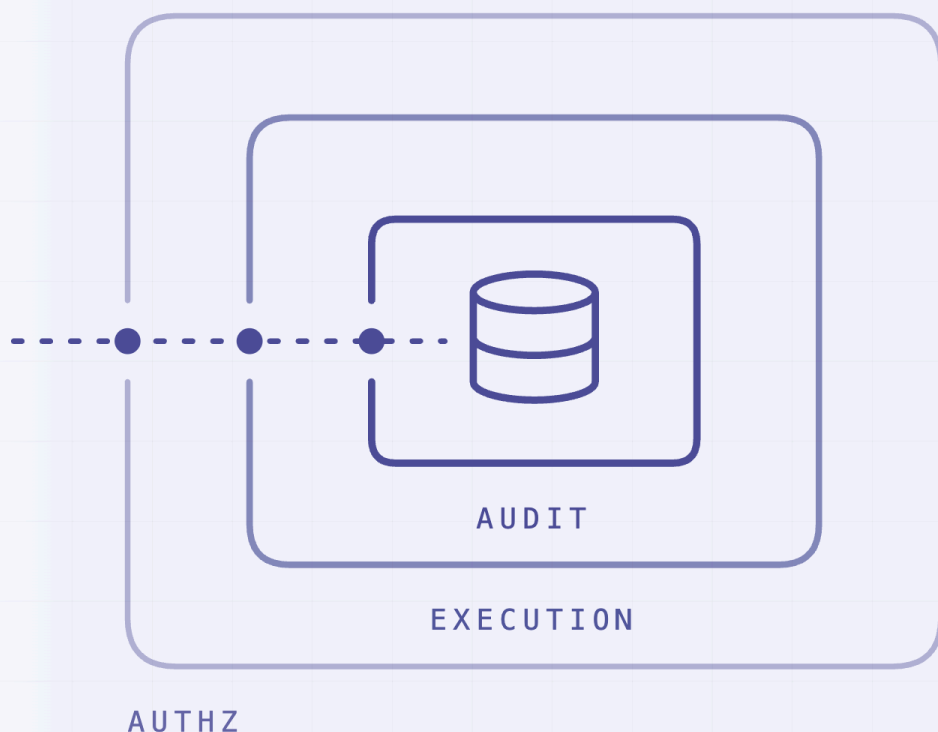




Guía de acceso a bases de datos en producción

Guía de acceso a bases de datos en producción

Identidades, credenciales, acceso just-in-time, ejecución mediada y auditoría

Muhammet Şafak

Índice

Prefacio	6
PARTE I — FUNDAMENTOS	13
Capítulo 1 — La puerta que nadie diseñó	14
Capítulo 2 — Un modelo de amenazas para los datos de producción	29
Capítulo 3 — Identidades, roles y mínimo privilegio	48
PARTE II — ACCESO DE LAS APLICACIONES	71
Capítulo 4 — Credenciales y secrets	72
Capítulo 5 — Las conexiones como política	91
Capítulo 6 — Lecturas, réplicas y copias de datos	110
PARTE III — ACCESO HUMANO	124
Capítulo 7 — Rutas de entrada: bastions, proxies y access gateways	125
Capítulo 8 — Acceso just-in-time y break-glass	148
Capítulo 9 — Ejecución mediada: análisis de consultas y operaciones peligrosas	175
Capítulo 10 — Consultas ad hoc y correcciones de datos	191
Capítulo 11 — Cambios de esquema y acceso para migraciones	217
PARTE IV — PROTEGER LOS DATOS	238
Capítulo 12 — Datos sensibles: clasificación, enmascaramiento y minimización	239
Capítulo 13 — Backups, exports y otras puertas laterales	262
PARTE V — DEMOSTRAR Y OPERAR	286
Capítulo 14 — Auditoría y rendición de cuentas	287
Capítulo 15 — El acceso durante los incidentes	306

Capítulo 16 — Una lista de verificación de acceso y reflexiones finales	324
Apéndice	339

Prefacio

Imagina un jueves por la tarde en Ledgerly, una empresa de facturación y pagos que no existe. Robin está on-call. La factura de un cliente lleva atascada en estado “sending” desde media tarde, y la corrección parece pequeña: una fila, una columna de estado. El cliente de base de datos de escritorio del equipo recuerda la contraseña de administrador compartida, se conecta al primary y hace commit automáticamente, como suelen hacer estos clientes. Una corrección de una sola fila pasa por ese cliente, bajo un rol que puede modificar todas las tablas, y su cláusula `WHERE` coincide con todo un lote de facturas en lugar de una.

Robin dedica el resto de la tarde a la reparación. Lo que lleva más tiempo es la pregunta que viene después: ¿quién ejecutó esa sentencia? El log de la base de datos muestra el rol de administrador compartido, que usan todos los servicios y, digamos, veinte personas. El gestor de contraseñas muestra quién abrió la entrada, no quién ejecutó qué. El equipo puede reconstruir el cambio a partir de los efectos secundarios en la aplicación, y al final lo hace. Pero durante un día más o menos, nadie puede decir con seguridad qué ocurrió exactamente, en qué filas, ni si algo más usó la misma puerta al mismo tiempo.

Nada en esta historia fue irrazonable por sí solo. La contraseña se compartió porque compartirla era la forma más rápida de poner en marcha el primer servicio. El cliente hacía commit automático porque ese es su comportamiento por defecto. El rol podía modificar todas las tablas porque, al principio, era más fácil que decidir qué tablas necesitaba cada servicio. Cada paso era razonable el día en que se dio. Juntos construyeron una puerta que nadie diseñó.

De esa puerta trata este libro.

Por qué existe este libro

El acceso a una base de datos de producción suele tratarse como fontanería. Se configura cuando se despliega el primer servicio y después solo se toca cuando algo se rompe o alguien se va. Mientras tanto, la base de datos se convierte en lo más valioso que opera la empresa: los clientes, el dinero, la historia. La distancia entre cuánto depende una empresa de ella y lo poco que se diseña su acceso es el tema de este libro.

Se ha escrito mucho sobre cada pieza. La documentación de las bases de datos explica cómo crear roles y grants. Las herramientas de secrets explican cómo emitir credenciales. Las guías de seguridad explican por qué el acceso permanente es un riesgo. Lo que cuesta más encontrar es el panorama completo desde el lado de la ingeniería: cómo encajan las piezas, qué le cuesta cada una a las personas que construyen y operan el sistema, y en qué orden podría adoptarlas un equipo con poco tiempo.

Este libro intenta ser ese panorama, y parte de una afirmación. El acceso a bases de datos en producción no es un problema de distribución de credenciales; es un problema de autorización, ejecución y rendición de cuentas.

La visión por defecto trata el acceso como una cuestión de quién tiene una contraseña y de cómo se guarda, se rota y se entrega. La tarde de jueves de arriba muestra dónde se queda corta esa visión. Robin tenía una credencial legítima, y la credencial hizo exactamente aquello para lo que se emitió. Lo que falló fue todo lo que la rodeaba: nada decidió que esa persona debía ejecutar una escritura contra esa tabla, nada inspeccionó la sentencia ni acotó su efecto antes del commit, y nada vinculó el resultado con una persona después. La autorización decide qué puede ocurrir y dónde. La ejecución es cómo llega cada sentencia a los datos. La rendición de cuentas es la evidencia que queda. Cada capítulo nombra el pilar al que sirve, y la Parte I construye el vocabulario de los tres. El libro trata cada control como una decisión con un precio. Un control que es adecuado para la tabla de tokens de pago puede carecer de sentido para una tabla de plantillas de factura, y el libro lo dirá cuando importe.

Debo ser claro sobre la postura. El libro está escrito desde el lado de quien defiende. Las amenazas se describen solo en la medida en que explican por qué existe un control; nada de lo que hay aquí es una guía para entrar.

No vende miedo y no cita estadísticas de brechas, porque una revisión de diseño serena es más útil que una alarmante. Y cuenta la fricción con honestidad. Cada control le pide algo a los ingenieros y a quien esté on-call, y un control que añade fricción sin reducir la exposición es teatro de seguridad. Cuando crea que una práctica popular es teatro, lo diré y explicaré por qué.

A quién va dirigido este libro

El libro está escrito para **ingenieros de backend y de plataforma** que construyen servicios que hablan con una base de datos de producción, para **roles de SRE y DBA** que operan esa base de datos, para **tech leads** que tienen que decidir cómo debe funcionar el acceso, y para **ingenieros que se han vuelto responsables de la seguridad hace poco** en un equipo que nunca tuvo una función dedicada. Si alguna vez te has preguntado si la contraseña compartida sigue siendo la misma contraseña compartida, has dudado antes de dar acceso de escritura a un colega, o te han preguntado “¿quién puede ver esta tabla?” y te has dado cuenta de que la respuesta honesta requiere un día de rastreo, eres a quien tengo en mente.

No necesitas formación en seguridad. Sí te conviene tener soltura con algunas cosas:

- Bases de datos relacionales: tablas, roles y grants, transacciones, y qué es una réplica.
- Servicios que se conectan a una base de datos a través de una red, y el vocabulario de los pools de conexiones y los timeouts.
- El trabajo on-call: recibir un page, reparar algo bajo presión y dejar por escrito después lo que pasó.

Conocer un motor de base de datos concreto ayuda con los ejemplos, pero no es imprescindible, como explica una sección posterior.

Lo que este libro no es

No es un libro de administración de bases de datos. No te enseñará a afinar un planificador, a dimensionar un buffer ni a elegir una topología de replicación. Donde la administración importa para el acceso, como con las

réplicas y las migraciones, el libro cubre esa parte y deja el resto a la documentación de tu propia base de datos.

No es un manual de cumplimiento. Regímenes como GDPR, KVKK, PCI DSS, SOC 2 y HIPAA aparecen porque determinan lo que preguntan los auditores y los clientes, y el libro los describe solo con ese espíritu: los regímenes de este tipo tienden a preguntar quién puede ver qué, cómo se aprueba el acceso y qué registro existe después. No interpreta ningún régimen, no cita sus artículos ni te dice qué se te exige. Esa interpretación corresponde a tus colegas de legal y de cumplimiento, y el objetivo del libro es darles una arquitectura sobre la que puedan razonar.

Y no es un libro de ataques. No encontrarás técnicas para entrar por la fuerza. Cuando un capítulo necesita una amenaza, nombra el riesgo en una frase, como una credencial que se filtra, una consulta que cae en la base de datos equivocada o una sesión de proveedor que queda abierta, y pasa al control. Si describir la amenaza con más precisión no cambiaría el diseño, el detalle se omite.

Cómo está organizado el libro

El libro tiene cinco partes y dieciséis capítulos, seguidos de un apéndice.

La Parte I, Fundamentos (capítulos 1 a 3) defiende la tesis anterior y construye el vocabulario para el resto. El Capítulo 1 examina cómo se acumulan las puertas hacia una base de datos. El Capítulo 2 construye un modelo de amenazas para los datos de producción que incluye tanto los errores como el mal uso. El Capítulo 3 trata las identidades, los roles y el mínimo privilegio, de los que dependen todos los capítulos posteriores.

La Parte II, Acceso de las aplicaciones (capítulos 4 a 6) trata de los servicios. El Capítulo 4 cubre las credenciales y los secrets, desde las contraseñas estáticas hasta las credenciales dinámicas y la workload identity. El Capítulo 5 trata las conexiones como política: pooling, límites y timeouts. El Capítulo 6 cubre las lecturas, las réplicas y las demás copias de datos que la producción crea sin hacer ruido.

La Parte III, Acceso humano (capítulos 7 a 11) trata de las personas. El Capítulo 7 compara los bastions, los proxies y los access gateways. El Capítulo 8 cubre el acceso just-in-time y el break-glass. El Capítulo 9 cubre la

ejecución en sí: cómo un gateway inspecciona cada sentencia, acota su efecto y la confirma con un commit. El Capítulo 10 cubre las consultas ad hoc y las correcciones de datos. El Capítulo 11 cubre los cambios de esquema y quién está autorizado a hacerlos.

La Parte IV, Proteger los datos (capítulos 12 y 13) pasa de quién puede conectarse a lo que puede ver. El Capítulo 12 cubre la clasificación, el enmascaramiento y la minimización. El Capítulo 13 cubre los backups, los exports y otras puertas laterales que esquivan todos los controles contruidos hasta ese punto.

La Parte V, Demostrar y operar (capítulos 14 a 16) cubre cómo sabe un equipo que sus controles funcionan y cómo los mantiene funcionando. El Capítulo 14 cubre los registros de auditoría y las revisiones de accesos. El Capítulo 15 cubre el acceso durante los incidentes, cuando la presión es mayor y los atajos son más tentadores. El Capítulo 16 ofrece una lista de verificación, una escalera de madurez con pruebas de salida y algunas reflexiones finales sobre la cultura.

El apéndice reúne un glosario, una referencia rápida de los patrones, una matriz de operaciones peligrosas, una correspondencia con requisitos de cumplimiento y plantillas para las revisiones de accesos y las revisiones posincidente.

Todos los capítulos tienen la misma forma. Abren con un escenario breve y abiertamente ficticio. Después explican el problema y por qué se queda corto el enfoque obvio, presentan el control, y exponen sus costos y cuándo no usarlo. Enumeran los errores comunes, tratados como trampas del diseño y nunca como defectos personales, y terminan con de cinco a siete ideas clave y un puente hacia el capítulo siguiente. La forma compartida significa que puedes leer los capítulos en orden o saltar al que corresponda al problema de esta semana, y aun así encontrar la sección de costos donde la esperas.

Conoce Ledgerly

Ledgerly es un servicio ficticio de facturación y pagos entre empresas. Las empresas emiten facturas a través de él, cobran pagos y llevan un libro mayor de lo que se movió. Funciona con cinco servicios: `billing-api`, `invoice-worker`, `payments-service`, `reporting-job` y `admin-console`.

Su base de datos `primary` tiene dos réplicas y una copia analítica para el equipo de datos. Almacena datos de contacto de clientes, información de cuentas bancarias y tokens de pago, y por eso importa el diseño de su acceso.

Las personas que la rodean cumplen roles conocidos. Hay ingenieros de `backend on-call`, un equipo de soporte, un equipo de datos, un pequeño grupo de plataforma que opera las bases de datos y un proveedor externo de contabilidad que recibe exports.

Al comienzo del libro, `Ledgerly` tiene una contraseña de administrador compartida, acceso permanente para la mayor parte del equipo de `backend`, clientes de escritorio en los laptops y ningún registro de quién hizo qué. Al final tiene un rol para cada servicio, credenciales de corta duración, un gateway para las personas, una forma revisada de corregir datos, sentencias que se comprueban y se acotan antes de ejecutarse, copias enmascaradas y un log que no se puede reescribir en silencio. Llega hasta ahí paso a paso, y cada capítulo dice qué paso añade y cuánto le cuesta al equipo.

`Ledgerly` es inventada, y también todo lo que le sucede. Cada capítulo abre con un escenario marcado como ficción, y cualquier cifra que aparece en él es hipotética. Nombres como Robin, Sam, Jordan y Alex son marcadores de posición de roles en un equipo. Cuando un escenario sale mal, la lección trata del diseño que permitió que saliera mal, y nunca de la persona de la historia.

Cómo leer el código

Ningún ejemplo de este libro está escrito en un lenguaje de programación real. Los procedimientos se escriben en pseudocódigo sencillo, y los roles, los grants y las tablas usan una notación parecida a SQL que no pertenece a ningún motor concreto. Nada del libro está pensado para pegarse en una terminal. No encontrarás sintaxis de configuración de un proxy, un vault o un producto de nube en particular, porque esa sintaxis cambia y el patrón que hay detrás no.

Los diagramas usan Mermaid. Cada uno va precedido de una frase que te dice qué mirar, y las rutas denegadas o fallidas están etiquetadas. Los ejemplos nunca contienen secrets, números de cuenta ni direcciones de

correo realistas; en su lugar aparecen marcadores como `<secret>` e `IBAN-PLACEHOLDER`.

Por qué en inglés y por qué independiente del motor

El libro está escrito en inglés porque la mayor parte del vocabulario de este campo, desde `least privilege` hasta `break-glass`, se usa en inglés incluso en equipos que trabajan en otros idiomas, y quise que los términos siguieran siendo reconocibles. El libro es independiente del motor por una razón relacionada. Un equipo puede usar PostgreSQL, MySQL u otra cosa, y puede usar un servicio de identidad en la nube, un gestor de secrets o un script hecho a mano. Los patrones son los mismos en todos ellos: separar al propietario de los usuarios, dar a cada workload su propia identidad, hacer que las credenciales venzan, poner delante de las personas una entrada consciente de la identidad, comprobar lo que se ejecuta antes de que se ejecute, hacer pasar las escrituras por algo que pueda rechazarlas, registrar lo que ocurre, revisarlo. Los productos y su sintaxis cambian a menudo. Cuando se nombra un producto real, se nombra como ejemplo, y su comportamiento se describe en términos que deberían valer en todas las versiones.

Una nota sobre la certeza

He intentado decir cuáles de mis recomendaciones son firmes y cuáles son preferencias. Algunas son firmes porque el costo de equivocarse es la exposición de datos de clientes o la pérdida del libro mayor, y lo digo cuando es así. La mayoría son juicios de valor, y un equipo reflexivo podría soportarlos de otra manera según su tamaño, sus reguladores, sus clientes y la frecuencia con que recibe pages. Donde no sé, diré “no lo sé” y nombraré las variables que resolverían la cuestión.

Si no estás de acuerdo con una recomendación, quiero que puedas encontrar la premisa con la que discrepas. Por eso cada capítulo detalla su razonamiento y sus costos. Un diseño que entiendes y que has adaptado a tus propias restricciones vale más que uno que copiaste de un libro.

El Capítulo 1 empieza por la puerta misma.

PARTE I

Fundamentos

Capítulo 1 — La puerta que nadie diseñó

Piensa en Ledgerly en sus primeros años. Un posible cliente envía un cuestionario de seguridad, y una línea parece rutinaria: “Enumere todas las personas y sistemas que pueden conectarse a su base de datos de producción”. Alex, que se ocupa de las bases de datos junto con otras tareas, lo recoge una tarde tranquila.

Alex empieza por el gestor de contraseñas, que guarda una sola entrada de administrador que conoce todo el equipo de backend. Después vienen los cinco servicios, cada uno con esa misma contraseña almacenada, y los laptops, donde los clientes de base de datos de escritorio la recuerdan. Luego la copia analítica, una restauración nocturna de un dump sin enmascarar, y unos cuantos dumps más antiguos en un almacenamiento compartido que nadie recuerda haber creado. Cada lugar que Alex revisa lleva a otro.

El cuestionario tiene una segunda línea: “¿Cómo saben lo que hicieron?”. Alex abre el log de la base de datos y encuentra el mismo nombre de rol en cada línea, ya venga la sentencia de un servicio, de un ingeniero que corrige una fila o de un informe que debería estar leyendo una réplica. La respuesta honesta a la primera pregunta es un montón de lugares, y la respuesta honesta a la segunda es que Ledgerly no lo sabe.

Alex no necesita una contraseña mejor. Alex necesita saber de qué está hecho el acceso y cómo sería una respuesta breve y defendible. Esa es la pregunta de este capítulo: ¿de qué está hecho el acceso a una base de datos de producción, y qué haría falta para que Ledgerly pudiera responder ese cuestionario sin una semana de rastreo?

Este capítulo describe el estado inicial de Ledgerly, igual que el Capítulo 2, y todavía no añade ningún paso del estado objetivo. Su tarea es darte el vocabulario y el mapa para los pasos que siguen.

El problema y por qué es difícil

El acceso a una base de datos de producción rara vez se diseña. Se acumula. El primer servicio necesita una forma de entrar, así que alguien crea un rol de administrador y una contraseña. El segundo servicio la reutiliza porque funcionó con el primero. Un ingeniero necesita mirar una fila, y la contraseña va a un cliente de escritorio. Un job programado necesita datos y recibe la misma cadena de conexión que todo lo demás. Ninguno de estos pasos es una decisión sobre el acceso, y la suma de todos ellos es la puerta que nombra el título del capítulo.

Mira cómo es la puerta de Ledgerly cuando Alex se pone a contar. La contraseña de administrador compartida está en un gestor de contraseñas del equipo, y supongamos, a modo de ilustración, que unas veinte personas la conocen. El rol de administrador que desbloquea es propietario de todas las tablas. Los cinco servicios se conectan con ese rol, con la misma contraseña guardada en la configuración de cada servicio, y el pipeline de deploy también ejecuta con ella las migraciones. Los ingenieros llegan al primary desde sus laptops a través de la VPN de la empresa, con clientes que recuerdan la contraseña y confirman cada sentencia en cuanto se ejecuta. El acceso es permanente: nadie lo pierde nunca, ni siquiera quienes se pasaron a otro equipo hace un año.

Las réplicas aceptan el mismo login de administrador, y las consultas de informes a veces se ejecutan en el primary. La copia analítica es una restauración de un dump sin enmascarar, y los archivos del dump se quedan ahí. Un proveedor externo de contabilidad nunca se conecta a ninguna base de datos de Ledgerly, pero recibe exports de datos de facturas y del libro mayor, y el paso de export de reporting-job escribe esos archivos con el login de administrador compartido. El log de la base de datos muestra un solo rol. Nada en este arreglo registra qué persona ejecutó qué sentencia.

Es tentador tratarlo como un problema de disciplina: decir a la gente que tenga cuidado, cambiar la contraseña con más frecuencia. No empezaría por ahí. Una regla que solo funciona cuando todos la recuerdan falla la primera vez que la persona on-call está cansada. El sistema da a cada principal las mismas capacidades y el mismo nombre, así que quien quiere ser

cuidadoso no tiene con qué: ningún rol más estrecho que elegir, ninguna forma de decirle “solo lectura” a la base de datos, ninguna sesión que lleve un nombre.

Lo mismo vale para el remedio al que se recurre primero, que es manejar mejor la contraseña: guardarla en un vault, rotarla, entregarla solo a quien la pida. Eso es distribución de credenciales, y vale la pena hacerlo, pero responde a una sola pregunta: quién tiene el secret. El acceso a bases de datos en producción no es un problema de distribución de credenciales; es un problema de autorización, ejecución y rendición de cuentas. La autorización pregunta qué puede hacer cada principal y por qué ruta. La ejecución pregunta cómo llega cada sentencia a los datos, si algo la inspecciona antes y hasta dónde puede alcanzar. La rendición de cuentas pregunta si la evidencia vincula cada sentencia con una persona y una solicitud. Una contraseña perfectamente guardada resuelve quién tiene el secret y nada más: quien la tiene aún puede ejecutar una sentencia que nadie inspeccionó, y el log aún puede mostrar un nombre compartido.

Por eso esta es una cuestión de ingeniería y no de política. Una política establece lo que debería ocurrir. Un diseño determina lo que puede ocurrir, y qué deja un registro cuando ocurre. El resto de este capítulo adopta la forma de un problema de diseño: de qué consta el acceso, qué queremos que haga y dónde se cubre cada parte en este libro.

El acceso tiene cinco partes

Antes de juzgar cualquier diseño, conviene tener una definición precisa de lo que se está diseñando. Me resulta útil decir que el acceso a una base de datos de producción tiene cinco partes: una identidad, un grant, una ruta, una ejecución y un registro. La tabla indica qué pregunta responde cada parte y a qué pilar de la tesis del libro pertenece.

Parte	Pregunta	Pilar
Identidad	¿Quién o qué actúa: una persona o una instancia de workload?	Rendición de cuentas
Grant	¿Qué puede hacer ese principal en absoluto: roles, privilegios, techo?	Autorización

Parte	Pregunta	Pilar
Ruta	¿Cómo llega la conexión: alcance de red, gateway, pooler?	Autorización (dónde)
Ejecución	¿Cómo llega una sentencia a los datos? Quién o qué la envía, si se inspecciona antes de ejecutarse, hasta dónde puede alcanzar (transacción, row guard, timeout) y cómo se confirma.	Ejecución
Registro	¿Qué evidencia queda, y vincula la sentencia con la persona, la solicitud y el resultado?	Rendición de cuentas

Los tres pilares se corresponden con las partes de este modo. La autorización es grant más ruta, decididos por solicitud. La ejecución es lo que le ocurre a cada sentencia. La rendición de cuentas es identidad más registro. Una contraseña toca solo la primera de ellas, y solo en la puerta.

La **identidad** es la afirmación verificada de que un principal es quien dice ser. Un **principal** es cualquier cosa que se autentica: una persona, un servicio, un job programado, una integración de un proveedor. Principal es la palabra que hay que usar cuando una afirmación vale por igual para personas y workloads, y este libro la usa así de principio a fin. Un rol no es una identidad. Un **rol** es un conjunto con nombre de privilegios dentro de la base de datos, y pueden compartirlo muchos principals, que es exactamente lo que hace el rol de administrador compartido.

El **grant** es lo que el principal tiene permitido hacer una vez dentro. Aquí un grant es el acto, y el registro resultante, de dar un privilegio a un rol, o un rol a un **login**; el Capítulo 3 amplía la definición. Un **privilegio** es una cosa concreta, como leer una tabla, y el login es la cuenta, a nivel de base de datos, con la que se conecta un principal. La **ruta** es el camino que sigue una conexión para llegar a la base de datos: una red, un proxy, un gateway, un laptop.

La **ejecución** es cómo llega una sentencia a los datos: quién o qué la envía, si algo la inspecciona antes de que se ejecute, qué acota su efecto y cómo se confirma. Un grant dice que un rol puede hacer un update sobre **invoices**. No dice que ese update deba tocar una sola fila, que alguien lo haya mirado antes ni que se ejecute dentro de una transacción que pueda deshacerse con un rollback. En el estado inicial nada responde a esas preguntas: el cliente de escritorio envía lo que se teclea y lo confirma en el acto. De esta parte trata el Capítulo 9.

El **registro** es la evidencia posterior, la respuesta a “¿quién hizo qué, y cuándo?” y, en el caso de una escritura, “¿en qué filas y bajo qué solicitud?”.

Cada parte puede fallar por sí sola, pero cada una depende de las demás para ser útil. Sin una identidad distinta, un grant preciso no tiene a quién asociarse. Sin una ruta elegida, una identidad puede esquivarse por un camino que nunca pide ninguna. Sin una ejecución controlada, un grant estrecho aún permite una sentencia más amplia de lo que nadie quiso. Sin un registro, no puedes saber si las otras cuatro funcionaron. Esta dependencia es la razón de que arreglar una sola parte, normalmente la contraseña, deje en pie el resto del problema.

Una prueba: ¿puede el log responder la pregunta?

Una manera de averiguar qué partes faltan es hacerle una pregunta al log y ver dónde se detiene la respuesta. El siguiente pseudocódigo sigue una sola sentencia a través de las cinco partes, usando solo lo que registra el estado inicial.

```
function explainStatement(logLine):
    login = logLine.login
    // identity: does this login map to exactly one principal?
    principals = principalsUsing(login)
    if count(principals) != 1:
        return { answer: "unknown", stoppedAt: "identity" }
    // path: did the connection arrive by a route we chose?
    if not isKnownPath(logLine.source):
        return { answer: "unknown", stoppedAt: "path" }
    // grant: was this statement within what the principal was meant
    to do?
    if not withinIntendedGrant(principals[0], logLine.statement):
        return { answer: principals[0], stoppedAt: "grant" }
    // execution: was a write reviewed and bounded before it
    committed?
    if isWrite(logLine.statement):
        if not logLine.reviewedStatement or not
logLine.countShownBeforeCommit:
        return { answer: principals[0], stoppedAt: "execution" }
        if logLine.rowsAffected is unknown or logLine.rowsAffected >
logLine.expectedRows:
        return { answer: principals[0], stoppedAt: "execution" }
    // record: does the evidence tie the statement to the person, the
    request and the result?
    if logLine.correlationId is empty or logLine.result is unknown:
        return { answer: principals[0], stoppedAt: "record" }
    return { answer: principals[0], stoppedAt: null }
```

En el estado inicial de Ledgerly, `principalsUsing("admin")` devuelve unas veinte personas, cinco servicios y el pipeline de deploy, así que la función se detiene en la primera comprobación para todas las sentencias. Eso no es un fallo del log. El log es veraz sobre lo que vio, que es un solo login. La brecha está en el diseño que puso a muchos principals detrás de un único login. Arregla la identidad, y la misma función se detendría a continuación en la comprobación de ejecución. El incidente fundacional es el caso para el que existe: una corrección de una sola fila cuya cláusula `WHERE` coincide con todo un lote tiene un `rowsAffected` muy superior a `expectedRows`, nunca se revisó y nunca mostró su conteo antes del `COMMIT`, así que se detiene en “execution” aunque la persona sea conocida y el grant lo permitiera. Una sentencia que supera esa comprobación aún puede detenerse en “record” si nada la vincula con una solicitud y un resultado.

Aplicaciones y personas, una sola pregunta

Cuando la gente oye “acceso a la base de datos”, suele imaginar ingenieros con herramientas de consulta. Esa imagen es incompleta de una manera que importa. Gran parte del tráfico hacia el primary de Ledgerly procede de software: `billing-api` atendiendo la aplicación web, `invoice-worker` enviando recordatorios, `payments-service` hablando con los proveedores de pago, `reporting-job` ejecutándose por la noche y `admin-console` actuando en nombre del personal de soporte. El software se conecta con más frecuencia, durante más tiempo y con menos supervisión que cualquier persona.

Los servicios y las personas fallan de maneras distintas, por eso la Parte II cubre las aplicaciones y la Parte III cubre a las personas. El contraste es concreto:

	Servicios	Personas
Frecuencia	Constante, las veinticuatro horas	Ocasional, a ráfagas
Supervisión	Ninguna mientras se ejecutan	Una persona vigila el resultado
Duración de la credencial	De corta duración y emitida por un broker (Capítulo 4)	Una sesión o una ventana aprobada (capítulos 7 y 8)

	Servicios	Personas
Fallo típico	Un secret filtrado, un pool de conexiones agotado, un deploy que apunta a la base de datos equivocada	Una sentencia más amplia de lo previsto, ejecutada a mano bajo presión

Ambos descansan sobre los fundamentos de la Parte I.

Hay un tipo más de principal que es fácil pasar por alto: un servicio que actúa en nombre de una persona. `admin-console` lo usa el personal de soporte, pero en el estado inicial se conecta con el login de administrador compartido como todo lo demás, así que la base de datos ve la consola, nunca a la persona que hizo clic. Sam, de soporte, mira un cliente cada vez, y el diseño deja que la consola alcance los datos de todos los clientes bajo un nombre que no es de nadie. El Capítulo 3 muestra cómo la consola pasa la identidad de la persona con cada acción, y el Capítulo 14 cómo se unen los registros de auditoría, para que la persona sobreviva al salto.

Seguridad y confiabilidad son la misma pregunta

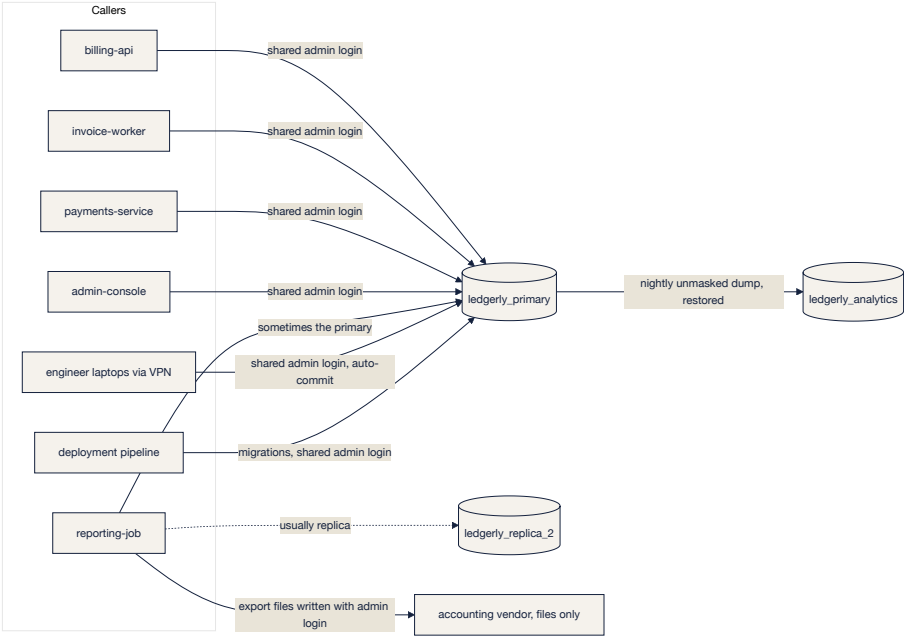
Los equipos suelen llevar dos listas. Una trata de seguridad: quién podría robar datos, quién podría abusar del acceso. La otra trata de operaciones: qué podría tumbar la base de datos o corromperla. Suelen ser responsabilidad de personas distintas y se revisan en momentos distintos. Para el acceso, las fusionaría, porque las preguntas son las mismas.

Toma tres eventos que suenan inconexos. Una sentencia con una cláusula `WHERE` equivocada o ausente actualiza todas las facturas en lugar de una. Un cliente de laptop deja una conexión abierta y ocupa un slot toda la tarde. Un dump de la base de datos de producción se copia a un almacenamiento compartido y lo lee alguien que no tenía nada que hacer leyéndolo. El primero es un problema de integridad de datos, el segundo uno de disponibilidad, el tercero uno de confidencialidad. Sin embargo, cada uno le pide lo mismo al diseño: ¿por qué un solo principal pudo hacer algo tan amplio, sin una ruta más estrecha, un límite o un registro?

Los tres eventos comparten una causa de diseño: ningún estrechamiento del grant, de la duración ni de la ruta, y nada que lo muestre después. La cláusula `WHERE` equivocada es sobre todo una brecha de grant y de ejecución, la conexión abierta una brecha de duración y de límites, y el dump

copiado una brecha de ruta. Estrechar el grant, limitar la duración, elegir la ruta, acotar la sentencia y conservar el registro ayudan con los tres eventos a la vez. Por eso un único diseño sirve a ambas listas. Un rol que no puede leer `payment_tokens` protege a los clientes de una filtración, y un rol que no puede escribir en `ledger_entries` protege el libro mayor de un update por lotes erróneo. Ambos nacen del mismo hábito de conceder de forma estrecha.

El diagrama siguiente muestra el estado inicial de Ledgerly. Fíjate en cuántas flechas terminan en el mismo nodo, y en la única flecha que sale de él hacia la copia analítica.



Dos cosas destacan. Primero, todas las rutas hacia el primary llevan el mismo nombre, así que la base de datos no distingue a un worker de una persona. Segundo, los datos también salen por una ruta lateral: la restauración nocturna copia todo, incluidas columnas que ningún analista necesita, a un almacén con su propia audiencia.

La gama de formas de entrar

Entre la contraseña compartida y un diseño totalmente controlado hay una gama de patrones, y los capítulos posteriores sitúan cada uno en ella. El diagrama va del menor al mayor control sobre cada sentencia. Fíjate en lo que añade cada paso y en dónde aparece la primera comprobación a nivel de sentencia.



Cada paso hacia la derecha añade una propiedad y cuesta algo. Un bastion estrecha la ruta, pero la sesión al otro lado sigue sin supervisión. Un session proxy vincula una conexión con una persona y la registra, pero entrega una sesión en la que se puede teclear cualquier cosa que el grant permita. Un gateway que media la ejecución mira cada sentencia antes de que se ejecute y puede rechazarla. Una ruta de producto elimina la sesión por completo: la persona pide una operación definida, como “reenviar esta factura”, y nunca escribe SQL. Nada de esto dice que el patrón de más a la derecha sea siempre el mejor; también es el que más hay que construir y el más lento de cambiar. El Capítulo 7 compara los cuatro primeros sobre rutas concretas, el Capítulo 9 cubre las comprobaciones a nivel de sentencia, y el capítulo final puntúa los patrones por pilar.

Cómo es un buen acceso

Escribiría el objetivo antes de elegir ninguna herramienta. Uso cinco propiedades, y cada una tiene un precio.

El **mínimo privilegio** es la práctica de conceder solo lo que una tarea necesita, durante el tiempo que lo necesita. El precio es el esfuerzo de averiguar qué necesita cada tarea, y la fricción cuando una tarea resulta necesitar más. Un equipo que lo adopta sin una forma rápida de pedir más verá a la gente esquivarlo.

La **vida corta** significa que el acceso termina por sí solo. Un grant que dura una hora no necesita que nadie lo recuerde. Contrasta esto con el **acceso permanente**, que existe de forma continua se use o no. Un grant permanente es la promesa de que nadie necesitará volver a pedir, y es có-

modo el primer mes y caro todos los meses posteriores, porque nadie recuerda qué cubría la promesa. El costo de la vida corta es pedir más veces, y un vencimiento que puede llegar a mitad de una reparación.

La **observabilidad** significa que cada uso del acceso deja un registro que nombra al principal, y que el registro está en un lugar que el principal no puede modificar. El costo es almacenamiento y, más importante, atención: un log que nadie lee es teatro de seguridad, ya que cuesta almacenamiento y no da protección. El Capítulo 14 vuelve sobre esto.

La **reversibilidad** tiene dos sentidos, y ambos importan. El acceso debería ser fácil de quitar, y un error cometido con acceso debería poder deshacerse o, al menos, contenerse. La revocación tiene que ser rutinaria, y deshacer un error requiere correcciones revisadas y backups que se restauren. Una forma barata de reversibilidad es ejecutar una escritura dentro de una transacción explícita y mostrar el conteo de filas antes del COMMIT: un update que debía tocar una fila y que informa de, digamos, cuatrocientas se detiene mientras todavía puede deshacerse con un rollback. El Capítulo 9 cubre cómo un gateway lo convierte en el comportamiento por defecto. El costo es el tiempo dedicado a los ensayos.

La **baja fricción** es la que los equipos olvidan anotar. Si el camino legítimo es lento, la gente encontrará el ilegítimo, y con razón, porque el incidente de las 2 de la madrugada no espera. El costo de la baja fricción es que limita a las otras cuatro: cada control tiene que ser lo bastante barato para usarse durante la peor hora de la semana.

Estas cinco tiran en direcciones opuestas. La vida corta añade fricción. La observabilidad añade un salto: una sesión que se registra tiene que pasar por algo que la registre, y ese algo puede ser lento o estar caído. El mínimo privilegio multiplica los roles. No creo que exista un ajuste que maximice las cinco. La pregunta útil no es “¿quién debería tener acceso?” sino “¿qué sentencias pueden ejecutarse, a través de qué y con qué registro?”. Cada capítulo posterior responde a una porción de ella, y dice lo que cuesta la respuesta.

Un primer inventario

Antes de adoptar cualquier control, anotaría lo que existe. El siguiente esbozo muestra la forma de un inventario para el estado inicial de Ledgerly. La idea es tener una fila por puerta, con un responsable con nombre y una razón, de modo que una fila sin responsable se vea como una brecha.

```
door:      desktop client on engineer laptops
principals: about twenty people (illustrative)
login:     shared admin
reaches:   ledgerly_primary, ledgerly_replica_1, ledgerly_replica_2
network source: company VPN, any laptop
grants:    owns every table
can write?: yes, auto-commit, no row limit
record:    database log shows the shared role only
owner:     nobody
reason:    "needed to fix rows during incidents"
ends:      never
```

Rellena un bloque por puerta: cada servicio, el job programado, el pipeline de deploy, los laptops, la restauración analítica, los dumps en el almacenamiento compartido y los archivos de export. La mayoría de los equipos descubre que las líneas `owner` y `ends` son las que quedan en blanco. Un inventario así no es un plan de proyecto. Es lo que convierte “¿quién puede llegar al primary?” de un día de rastreo en un documento que puedes leer.

Un mapa de este libro

Los capítulos siguientes toman por turno cada parte del acceso y cada fallo anterior. La tabla vincula cada pregunta que planteó este capítulo con el lugar donde el libro la responde.

La pregunta	Pilar	Dónde se responde
¿De quién y de qué protegemos los datos, y frente a qué tipos de eventos?	Los tres	Capítulo 2
¿Cómo damos a cada principal una identidad distinta y el rol más estrecho que funcione?	Autorización	Capítulo 3
¿Cómo obtienen los servicios credenciales sin una contraseña compartida en un archivo?	Rendición de cuentas (identidad)	Capítulo 4

La pregunta	Pilar	Dónde se responde
¿Cómo se convierten los límites de conexiones y los timeouts en una política que protege la base de datos?	Autorización y ejecución (límites)	Capítulo 5
¿Cómo llegan las lecturas a las réplicas, y qué pasa con las demás copias de datos?	Autorización	Capítulo 6
¿Por qué rutas llegan las personas a la base de datos, y qué hay en cada ruta?	Autorización (ruta), rendición de cuentas	Capítulo 7
¿Cómo obtiene alguien acceso de escritura para una tarea, y cómo termina?	Autorización	Capítulo 8
¿Cómo se comprueba una sentencia antes de ejecutarla, y cómo se confirma?	Ejecución	Capítulo 9
¿Cómo se mantienen revisables las consultas ad hoc y las correcciones de datos?	Ejecución	Capítulo 10
¿Quién cambia el esquema, y con qué acceso?	Ejecución	Capítulo 11
¿Qué datos necesitan protección adicional, y cómo se enmascaran o se retienen?	Autorización	Capítulo 12
¿Qué pasa con los backups, los exports y las demás vías que rodean los controles?	Autorización (ruta)	Capítulo 13
¿Cómo conservamos un registro que resista, y cómo lo revisamos?	Rendición de cuentas	Capítulo 14
¿Qué cambia durante un incidente?	Los tres, bajo presión	Capítulo 15
¿Qué hacemos primero, y cómo lo mantenemos?	Los tres	Capítulo 16

El libro tiene cinco partes: fundamentos, acceso de las aplicaciones, acceso humano, protección de los datos y sus copias, y demostrar y operar todo ello. Cada capítulo nombra al comienzo el pilar al que sirve. No tienes que leer en orden. Si tu problema inmediato es la contraseña compartida en cinco archivos de configuración, el Capítulo 4 es el lugar para empezar, pero el Capítulo 3 te evitará crear sin querer el siguiente rol compartido.

Trade-offs y costos

El primer costo es **el tiempo y la atención de un grupo pequeño**. Alguien tiene que construir el inventario, definir los roles, configurar las credenciales y llevar las revisiones. En un equipo pequeño, suele ser el traba-

jo a tiempo parcial de una sola persona, y compite con el trabajo de producto. Si el trabajo nunca se programa, el diseño se queda en el papel. Prefiero ver un primer paso pequeño y terminado que un plan grande.

El segundo costo es **la fricción para quien está on-call**. Cada control pone un paso entre una persona cansada y una reparación. Los roles estrechos significan que el primer intento a veces falla con un error de privilegios en el peor momento. El acceso de corta duración significa volver a pedirlo. Decide de antemano cuál es el camino rápido, y ensáyalo. Un control que se esquiva durante cada incidente no es un control; es un retraso antes del atajo.

El tercer costo son **nuevos sistemas que tienen que estar disponibles cuando se necesitan**. Un gateway, un almacén de secrets y un almacén de logs son, cada uno, algo más que puede caerse. Antes de adoptar uno, pregunta qué le ocurre a la persona que necesita la base de datos durante esa caída, y ten una respuesta preparada.

También hay casos en los que el diseño completo es el lugar equivocado para empezar. Un prototipo de un solo servicio sin datos de clientes, llevado por dos personas, no necesita un gateway ni un calendario de revisiones. Sí necesita su propio login, separado del de cualquier persona, porque es barato, y una nota que diga qué haría que el equipo volviera sobre la cuestión: el primer cliente, el tercer ingeniero o la primera integración externa. No sé cuál es el momento adecuado para tu empresa. Depende, y aquí está de qué: de lo sensibles que sean los datos, de cuántos principals los toquen y de la rapidez con que una sentencia errónea pueda causar un daño que no puedas revertir.

Por último, las propiedades del buen acceso tienen un límite. Reducen el radio de impacto y hacen visible el comportamiento. No hacen segura una base de datos, y ningún capítulo de este libro afirma lo contrario. El **radio de impacto** es cuánto puede verse afectado si un principal, una credencial o una acción sale mal, y el trabajo de este libro es hacer ese radio más pequeño y más visible, no hacerlo cero.

Errores comunes

Son trampas de diseño, no defectos personales. Cada una es tentadora por una razón, y cada una tiene un remedio que no depende de que nadie sea más cuidadoso.

- **Tratar el cambio de contraseña como el remedio.** Una contraseña nueva parece un avance, y elimina el conocimiento de un antiguo colega. Pero si, digamos, veinte personas y cinco servicios la siguen compartiendo, el diseño no ha cambiado. La solución es un diseño en el que nadie necesite compartir una contraseña, que es lo que construyen los capítulos 3 y 4.
- **Arreglar el grant y olvidar la ruta.** Los roles estrechos ayudan solo si las conexiones tienen que usarlos. Si los laptops aún pueden llegar al primary con un antiguo login de administrador, los roles estrechos son opcionales. Inventaría todas las rutas, no solo todos los roles.
- **Dejar fuera las aplicaciones.** Las revisiones de “quién tiene acceso a la base de datos” a menudo enumeran solo a personas, porque las personas aparecen en un organigrama. Los servicios tienen las credenciales más capaces y las usan más. Incluye cada workload, cada job programado y cada script.
- **Hacer el camino seguro más lento que el inseguro.** Si leer producción requiere un ticket y un día mientras que la contraseña compartida requiere, digamos, diez segundos, la contraseña compartida gana siempre. Apunta a un camino seguro que no sea más lento.

Advertencia: Un dump sin enmascarar en un almacenamiento compartido es la vía más directa del estado inicial hacia los datos de los clientes, y está abierta a cualquiera que pueda leer la carpeta. Considera que encontrar y eliminar estos archivos es parte del primer inventario, antes de cualquier rediseño de roles.

Ideas clave

- El acceso a bases de datos en producción no es un problema de distribución de credenciales; es un problema de autorización, ejecución y rendición de cuentas, y una contraseña responde solo a la primera pregunta, en la puerta. El acceso es un diseño de cinco partes, a saber,

Identidad, Grant, Ruta, Ejecución y Registro, y una debilidad en cualquiera de ellas socava a las demás.

- Una cuenta de administrador compartida no es una decisión que alguien tomó; se acumula, y el remedio es un diseño en el que nadie necesite compartir una contraseña.
- Las aplicaciones se conectan con más frecuencia y con menos supervisión que las personas, así que cualquier tratamiento del acceso tiene que cubrir servicios, jobs y herramientas como principals junto a los humanos.
- La seguridad y la confiabilidad operativa son una sola pregunta para el acceso, porque una cláusula `WHERE` equivocada, una conexión filtrada y un dump copiado comparten una causa de diseño: ningún estrechamiento del grant, de la duración ni de la ruta, y ningún registro.
- El buen acceso tiene cinco propiedades, que son mínimo privilegio, vida corta, observabilidad, reversibilidad y baja fricción, y tiran en direcciones opuestas, así que cada control debería presentarse con lo que cuesta.
- El primer paso es un inventario con una fila por puerta, que incluya los servicios, los laptops y las copias de datos, donde un responsable ausente o una fecha de fin ausente se ven como una brecha visible.
- Los controles reducen el radio de impacto y hacen visible el comportamiento, pero no hacen segura una base de datos, y el objetivo honesto es un radio más pequeño y una vista más clara.

El capítulo siguiente pasa de la forma del acceso a las razones para protegerlo. El Capítulo 2 pregunta de quién y de qué son los controles, clasifica en tres tipos los eventos que pueden salir mal y escribe un modelo de amenazas para Ledgerly al que pueda remitirse el resto del libro.

Capítulo 2 — Un modelo de amenazas para los datos de producción

Imagina que el contacto de seguridad de Ledgerly le envía al equipo de plataforma una pregunta breve antes de una revisión trimestral: “¿De quién protegemos la base de datos de producción?”. Alex, quien administra las bases de datos, responde en menos de un minuto: de gente externa, sobre todo. Luego Alex se detiene, porque la respuesta no coincide con los tickets del propio equipo del último año. Una solicitud de soporte que tocó al cliente equivocado. Una consulta de informes que ralentizó la escritura de facturas durante una tarde. Un dump restaurado en un laptop durante un día de depuración, y nadie recordó borrarlo. Un export para un proveedor que salió con una columna más de la prevista.

Ninguno de esos casos involucró a alguien externo. Ninguno tuvo mala intención. En todos hubo una puerta abierta a alguien que tenía una tarea razonable y un rol que permitía más de lo que la tarea necesitaba. La primera respuesta de Alex fue la que suena a seguridad. La segunda lista es la que describe el sistema.

Este capítulo describe Ledgerly en su estado inicial, como hizo el Capítulo 1, y no añade ningún control propio. Lo que añade es un modelo de amenazas escrito, la referencia frente a la cual se elige cada control posterior del libro. Plantea las amenazas frente a los tres pilares: qué puede hacer un principal, cómo llega cada sentencia a los datos y qué evidencia queda. La pregunta que responde es la que Alex no pudo contestar en un minuto: ¿quién y qué puede dañar los datos, por qué puertas y hasta dónde llega el daño?

El problema y por qué es difícil

El diseño de accesos sin un modelo de amenazas tiende a seguir la preocupación más ruidosa. Si es un atacante externo, el equipo construye un perímetro: una VPN, una regla de firewall, una allowlist de red. Eso importa. Pero un perímetro responde a “¿quién puede acercarse a la base de datos?” y no dice nada sobre lo que una persona o un servicio puede hacer una vez dentro. En el estado inicial de Ledgerly, cualquiera que esté en la VPN con la contraseña compartida puede hacer todo lo que hace el rol admin, es decir, todo.

La alternativa obvia es una lista de controles sacada de una lista de verificación y aplicada en orden. Una lista de verificación no sabe cuáles de tus puertas concentran más riesgo. Pedirá rotar una contraseña que nadie podría rotar sin una caída, y no dirá nada sobre la restauración nocturna que copia cada columna Restricted a un almacén con una audiencia mucho más amplia. Los controles elegidos sin un modelo se eligen por disponibilidad y por moda. Algunos reducen la exposición. Otros son teatro de seguridad: fricción sin una reducción equivalente del riesgo, y el equipo no sabe distinguirlos porque nunca escribió para qué servía cada control.

Hace falta un modelo escrito por tres razones. Primero, las puertas son objetos técnicos: un login, una ruta de red, un archivo en un almacenamiento compartido, una variable de pipeline. Una política puede decir “el acceso debe ser adecuado”, pero solo un diseño decide qué puede alcanzar un login. Segundo, los mayores riesgos nacen de la interacción de decisiones corrientes, como describe el Prefacio: una contraseña compartida, un cliente con autocommit y un rol que es propietario de todas las tablas. Ninguna decisión suelta parece mala, así que una revisión de decisiones sueltas no encontrará el problema. Tercero, el modelo tiene que ser lo bastante barato de mantener, o se escribirá una vez y se dejará a la deriva.

Un modelo de amenazas, en el sentido que se usa aquí, es pequeño. Enumera quién y qué puede actuar sobre los datos, por qué ruta, qué podría salir mal, hasta dónde llega el efecto y qué control lo atiende. No es una puntuación de riesgo ni una predicción. Es una manera de hacer la misma pregunta sobre cada puerta, de modo que las respuestas se puedan comparar.

Nota: El libro describe las amenazas solo en la medida en que justifican un control. Este capítulo nombra lo que puede salir mal, como que se filtre una credencial o que una consulta caiga en la base de datos equivocada, y no describe cómo lo provocaría alguien. Si más detalle no cambiara el diseño, se omite.

Quién y qué llega a la base de datos

El Capítulo 1 dijo que el acceso a una base de datos de producción tiene cinco partes: una identidad, un grant, una ruta, una ejecución y un registro. Un modelo de amenazas empieza por enumerar los principales y les hace las cinco preguntas a cada uno. Ledgerly tiene nueve tipos de actor que vale la pena enumerar. Dos de ellos, el proveedor y el atacante externo, no son principales en sentido estricto, y por eso mismo pertenecen a la lista: es fácil dejarlos fuera.

Servicios

Los servicios (`billing-api`, `invoice-worker`, `payments-service` y `admin-console`, con `reporting-job` tratado a continuación como un job) son los mayores usuarios de la base de datos. Funcionan sin supervisión, guardan sus credenciales en la configuración y solo se comportan tan bien como su código. Un servicio puede portarse mal sin que nadie haga nada incorrecto: un bucle de retries que abre conexiones hasta alcanzar el límite, una consulta que recorre una tabla entera porque se eliminó un índice, un deploy que apunta una build de staging al primary.

En el estado inicial todos los servicios se conectan con el mismo login admin, así que el radio de impacto de cada servicio es la base de datos completa, aunque `billing-api` e `invoice-worker` necesiten solo un puñado de tablas cada uno. Un defecto en el servicio más pequeño tiene el alcance del más grande.

Jobs por lotes

`reporting-job` merece su propia línea porque un job se ejecuta según un calendario, a menudo de noche, sin nadie mirando. Suele leer muchos datos, mantener transacciones largas y producir archivos. En el estado inicial a veces lee del primary, y su paso de export escribe los archivos para el

proveedor con el login admin compartido. Su comportamiento normal ya parece un acceso masivo, lo que hace más difícil detectar su comportamiento anómalo que el de una persona.

Ingeniería on-call

La rotación on-call es la que soporta más presión para tener acceso amplio, y con razón: un page de madrugada no es el momento de esperar una aprobación que nadie está despierto para dar. En el estado inicial usan la contraseña admin compartida a través de clientes de escritorio que la recuerdan y hacen commit en cada sentencia. La combinación de urgencia por arreglar y ausencia de fricción para cambiar cualquier cosa convierte a este en el grupo donde los accidentes son más probables y el que el diseño puede proteger mejor.

Soporte

El personal de soporte usa `admin-console` para mirar un cliente a la vez. La amenaza aquí es el alcance, no la malicia. Una herramienta capaz de mostrar a todos los clientes, respaldada por un rol que puede leer todas las columnas, responde a una pregunta sobre una factura con un sistema que podría mostrar todas las facturas. La pregunta de diseño es cómo hacer que la capacidad amplia no esté disponible cuando solo hace falta la estrecha.

El equipo de datos

El equipo de datos trabaja sobre `ledgerly_analytics`, que en el estado inicial es una restauración nocturna de un dump sin enmascarar: todas las columnas Restricted y Confidential, a las que accede un grupo cuyo trabajo es mirar datos en bloque. Su acceso es legítimo. Lo preocupante es que la copia tiene una audiencia más amplia, un manejo más laxo y menos logs que el primary, de modo que un control sobre el primary no protege lo que hay en la copia.

Proveedores e integraciones

El proveedor externo de contabilidad nunca se conecta a una base de datos de Ledgerly. Recibe archivos. Eso cambia la forma de la amenaza: la exposición no es un login sino un export, y su radio de impacto es el con-

junto de columnas que contiene el export y el tiempo que la copia exista del lado del proveedor. Las preguntas relevantes son qué contiene el archivo, quién es responsable de él después y si alguien notaría que el archivo ganó una columna.

CI/CD

El pipeline de despliegue también es un principal, aunque sea infraestructura y no una persona ni un servicio. En el estado inicial ejecuta las migraciones con el mismo login admin, porque el rol admin es propietario de todas las tablas y nada más puede cambiarlas, y la credencial reside en la configuración del pipeline. Un pipeline que puede cambiar cualquier tabla y guarda una credencial permanente es a la vez un riesgo de compromiso y un riesgo de error, ya que una migración defectuosa se ejecuta con todos los privilegios del propietario.

Personal de plataforma y DBA

El pequeño grupo de plataforma administra las bases de datos, y es el único actor que puede cambiar los controles mismos: los roles, los grants, la entrada por la que pasan las personas, el log. En el estado inicial usa el login admin compartido como todos los demás, así que nada separa su trabajo del de cualquiera. En el diseño objetivo tiene un rol para gestionar el acceso, `ledgerly_access_admin`, que tres principales usan a través de sus propios logins. Eso no elimina la amenaza; la traslada. Un grant equivocado o un ajuste de log desactivado por alguien con este rol cambia el significado de todas las demás filas del modelo.

Por eso el modelo enumera el **plano de control** como un activo por derecho propio: el access gateway, el credential broker, `ledgerly_access_admin` y el almacén de auditoría. Los datos los protegen los controles, y los controles los protege un conjunto mucho más pequeño de personas y sistemas. La exposición es pequeña en volumen y grande en efecto, y las preguntas son las mismas que para cualquier puerta: quién puede alcanzarla, qué podría cambiar y quién lo notaría.

El atacante externo

Un atacante externo no es un principal hasta que obtiene una credencial, así que la forma útil de modelarlo es por lo que necesitaría obtener, no por lo que podría hacer. Las preguntas de quien defiende son: qué puertas aceptan una credencial que podría filtrarse, qué alcanzaría y cómo nos enteraríamos. Las respuestas salen de los otros ocho actores. Una credencial filtrada hereda el radio de impacto de a quien perteneció.

Tres clases de amenaza

Los mismos actores pueden causar daño de tres maneras distintas, y los controles difieren en cada una. Mezclar las clases lleva a controles que no corresponden: una aprobación de la intención, diseñada contra el mal uso, hace poco contra un accidente, porque quien aprueba está de acuerdo en que la tarea es legítima y la sentencia sigue saliendo mal.

Errores

Un error es una acción que debía ser correcta y no lo fue: una sentencia ejecutada contra el entorno equivocado, una condición que coincide con más filas de las previstas, una consulta demasiado pesada para el primary, un script apuntado a producción por una configuración obsoleta. El actor es un principal que hace su trabajo.

Yo esperarí que esta clase sea la más frecuente en la mayoría de los equipos, por una razón estructural y no estadística. El mal uso necesita a una persona con un motivo, y el compromiso necesita a un atacante con una oportunidad, pero un error solo necesita a una persona con una tarea. No tengo ninguna cifra de cuánto más frecuentes son los errores, y no confiaría en una si me la ofrecieran. La conclusión para el diseño se sostiene de todos modos: los controles dirigidos a los errores protegen contra la clase que todo equipo tiene, se enfrente o no alguna vez a las otras dos.

Los errores se atienden mejor estrechando lo que puede alcanzar una sola sentencia, ya que no se puede pedir a las personas que tengan cuidado para siempre. Pertenecen a esta clase los roles que no pueden tocar tablas que no necesitan (Capítulo 3), los timeouts que detienen una consulta descontrolada (Capítulo 5), el análisis de cada sentencia antes de ejecutar-

la (Capítulo 9), una ruta distinta para las escrituras con revisión four-eyes (capítulos 8 y 10) y una forma revisada de reparar los datos después (Capítulo 10).

El paso four-eyes ayuda aquí solo con una condición: que quien revisa vea la sentencia y su efecto esperado. Aprobar la intención (“corregir la factura 1042”) es un control para la clase del mal uso y deja intacta la clase de los errores. Aprobar la sentencia y el número de filas que debería cambiar atiende los errores.

Mal uso y curiosidad

Mal uso significa aquí que un principal con acceso legítimo lo usa para algo ajeno a la tarea: leer el registro de un cliente por interés, exportar más de lo que el trabajo necesita, mirar los datos de colegas. Incluye la simple curiosidad, que probablemente es la forma más común y la más difícil de calificar como falta. Incluye también el caso deliberado, que es raro.

La respuesta de diseño no es la sospecha hacia el equipo. Un equipo no necesita suponer mala fe para notar que un grant permanente hace posible curiosear y no deja rastro. Los controles que importan mantienen el acceso atado a una tarea y lo hacen visible: roles de solo lectura que retienen las columnas Restricted, vistas enmascaradas de los datos Confidential (Capítulo 12), grants con límite de tiempo (Capítulo 8) y un registro de auditoría que quien revisa realmente lee (Capítulo 14). Sin evidencia no se puede responder “¿fue apropiado este acceso?” para nadie, incluidas las personas que no hicieron nada mal.

Compromiso

Compromiso significa que una credencial o una máquina está en manos de alguien que no es el principal previsto: una contraseña que salió por un archivo de configuración, un laptop que se pierde, una variable de pipeline leída por algo en lo que no debería confiarse. Quien defiende no necesita saber cómo ocurrió para diseñar contando con que puede ocurrir. El diseño controla qué podría hacer la credencial, durante cuánto tiempo y si alguien se enteraría.

El compromiso es donde más importan las propiedades de una credencial. Una credencial compartida, estática y sin restricciones convierte la pérdida de un laptop en la pérdida de todas las tablas. Una credencial limitada a un rol, de vida corta y emitida para una sesión convierte el mismo suceso en uno acotado y visible. Los capítulos 4 y 7 tratan cómo se emiten las credenciales; el Capítulo 3, cuán poco deben llevar; el Capítulo 14, cómo darse cuenta.

Nota: Clasificar un suceso suele ser cuestión de criterio. Una corrección de una sola fila que alcanza a todo un lote es un error. La misma sentencia ejecutada a propósito por alguien sin ninguna tarea es mal uso, y ejecutada con una credencial tomada de un laptop es compromiso. La base de datos ve la misma sentencia en cada caso, y ese es el argumento a favor de logs que nombren un login y de controles que no dependan de adivinar la intención.

Qué se protege

Vale la pena proteger tres propiedades de los datos, y el modelo debería decir cuál toca cada amenaza. Son la conocida tríada de confidencialidad, integridad y disponibilidad, y cada una tiene un significado propio en Ledgerly.

La confidencialidad trata de quién puede leer qué. En Ledgerly el peso es desigual. `iban`, `account_holder_name` y `token_value` son Restricted, de modo que su exposición sería difícil de deshacer. `contact_name`, `contact_email`, `contact_phone` y `billing_address` son datos PII de clientes Confidential (información de identificación personal). Los importes de `invoices` y `payments` son Internal. Una amenaza que toca datos Internal merece menos esfuerzo de diseño que una que toca datos Restricted, y el modelo debería mostrar esa diferencia en lugar de tratar todas las columnas por igual. Los regímenes de protección de datos de este tipo suelen preguntar qué datos personales tienes y quién puede verlos, que es la misma diferencia; la interpretación corresponde a tus colegas de legal y de cumplimiento.

La integridad trata de si los datos siguen siendo correctos. En la mayoría de las tablas un error puede corregirse con una escritura posterior, pero `ledger_entries` y `audit_events` son append-only, y para el ledger la integridad importa más que el secreto: una entrada equivocada se arregla con una nueva entrada correctiva a través de `payments-service`, nunca con una edición. Una amenaza a la integridad de esas dos tablas es por tanto distinta en naturaleza. Es también aquella en la que la evidencia importa más, ya que un ledger que puede cambiarse en silencio es un ledger en el que nadie puede confiar.

La disponibilidad trata de si la base de datos atiende a los servicios que dependen de ella. Es la propiedad que más se olvida en las discusiones de seguridad, y aquella en la que dominan los accidentes. Un bloqueo largo, una consulta que recorre una tabla grande en el primary o una avalancha de conexiones desde un bucle de retries pueden detener la facturación y el cobro de pagos tan por completo como podría hacerlo un ataque. La disponibilidad tiene filas de primera clase en el modelo, ligadas a la política de conexiones (Capítulo 5) y a adónde se permite que vayan las lecturas (Capítulo 6).

Una cuarta propiedad acompaña a estas tres: la **rendición de cuentas**, es decir, que el registro pueda decir qué principal hizo qué. Es la parte de registro del acceso. En el estado inicial falta, pues el log solo muestra el rol compartido, y esa ausencia eleva el costo de todas las demás amenazas.

Radio de impacto

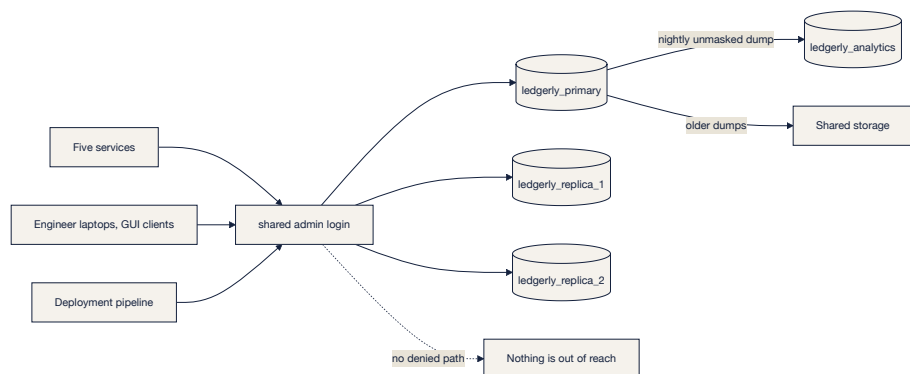
El modelo necesita una herramienta más para comparar amenazas: una forma de decir hasta dónde llega un solo fallo. El **radio de impacto** es cuánto puede verse afectado si un principal, una credencial o una acción sale mal. El término solo es útil cuando se usa sobre algo concreto, así que la pregunta siempre es “¿cuál es el radio de impacto de este login?” y nunca “¿cuál es nuestro radio de impacto?”.

Un radio de impacto tiene siete dimensiones, y ayuda mirar cada una por separado:

- **Alcance de datos:** qué tablas, columnas y filas puede tocar el principal.

- **Alcance de acciones:** si puede leer, escribir, borrar o cambiar el esquema.
- **Alcance de entornos:** a qué bases de datos puede conectarse: el primary, las réplicas, la copia de analítica.
- **Alcance temporal:** cuánto tiempo sigue siendo válido el acceso, desde una sesión hasta siempre.
- **Alcance de copias:** dónde más se han duplicado los datos que puede leer y quién puede acceder a esas copias.
- **Alcance de sentencias:** cuánto puede cambiar una sola sentencia antes de que algo la detenga: las filas que toca, si se ejecuta en una transacción que se puede revertir y si un timeout la acota.
- **Origen de red:** qué redes y máquinas pueden presentar la credencial.

El login admin compartido del estado inicial de Ledgerly lleva todas las dimensiones al máximo. Alcanza todas las tablas y todas las columnas, puede leer, escribir, borrar y alterar, se conecta al primary y a ambas réplicas, no vence y sus datos ya tienen copias fuera del primary. Cualquier sentencia no tiene límite y se confirma de inmediato, y cualquier laptop en la VPN puede presentar la credencial. El diagrama muestra qué mirar: todo lo que está a la izquierda tiene la misma credencial, así que todo lo que está a la derecha queda dentro de un solo radio.



Compáralo con el rol del estado objetivo `app_reporting_job`: solo lectura, sobre una réplica, sin acceso a columnas Restricted. Un defecto que en el estado inicial podría exponer datos Restricted es, para ese rol, una consulta pesada sobre una réplica. El defecto es idéntico; el radio de impacto cambió.

Para que la comparación sea honesta, escribe el radio por login, como una lista de preguntas y no como una puntuación, y pon dos logins lado a lado. El esquema siguiente es ilustrativo, una ayuda de revisión y no una herramienta sobre la que se apoye el libro. Una respuesta desconocida se anota como desconocida, porque un modelo que redondea lo desconocido a “bien” oculta los casos para los que existe.

dimension	shared admin login	app_reporting_job
data reach	every table	unknown until Chapter 3
lists the tables		
restricted cols	all	none
action reach	read, write, delete, DDL	read
environment	primary, both replicas	one replica
time reach	no expiry	unknown until Chapter 4
decides		
copy reach	unmasked restore, dumps	the export files it
writes		
statement reach	unbounded, commits at once	read-only transaction,
statement timeout		
network origin	company VPN, any laptop	unknown until Chapter 7
decides		

La comparación es útil porque es desigual. Donde la segunda columna dice “unknown”, el modelo ha encontrado una pregunta que el diseño aún no ha respondido, y eso es un hallazgo y no un hueco en la tabla.

Se aplican dos límites. Un radio describe lo que un login podría hacer, no lo que hará, de modo que es un límite de planificación y no un pronóstico. Y no dice nada sobre cuán probable es cada amenaza. Un radio estrecho en una puerta abierta todo el día puede merecer más atención que un radio amplio en una puerta que se abre una vez por trimestre, y por eso el modelo tiene que combinar el radio con la exposición.

El modelo de amenazas de Ledgerly

Con los actores, las clases, las propiedades y el radio de impacto ya establecidos, el modelo cabe en una sola tabla. La tabla siguiente cubre el estado inicial de Ledgerly. Cada fila es un actor y una ruta, con la clase de amenaza que más importa para ese par, el impacto sobre las propiedades anteriores, los capítulos que aportan el control y el **riesgo residual**: lo que sigue siendo cierto cuando esos controles están en su sitio. La “exposición”

es un juicio cualitativo (alta, media o baja) de la frecuencia con que se usa la puerta y de lo fácil que es usarla mal, y se deja por escrito para que alguien pueda discrepar de él.

Actor	Ruta	Clase principal	Impacto	Exposición	Se controla en	Riesgo residual
Servicios (los cinco)	Login admin compartido desde la configuración de cada servicio	Error, compromiso	Cualquier tabla, todas las columnas; disponibilidad si se agotan las conexiones	Alta	capítulos 3, 4, 5	Un defecto aún tiene el alcance de su propio rol
reporting-job	Lee el primary en algunas ejecuciones	Error	Las lecturas pesadas compiten con las escrituras; disponibilidad del primary	Media	capítulos 5, 6	Una lectura pesada en una réplica aún ralentiza a otros lectores de la réplica
Ingeniería on-call	Laptop, cliente GUI, contraseña admin compartida, autocommit	Error, compromiso	Escrituras sin alcance acotado; integridad de cualquier tabla	Alta	capítulos 7, 8, 10	Una sentencia escrita a mano dentro de un rol concedido aún puede ser errónea
Soporte	admin-console actuando bajo el login compartido	Mal uso, error	Curioso de datos de clientes Confidential; ediciones sobre el cliente equivocado	Media	capítulos 3, 10, 12	Soporte aún ve las columnas que necesita el caso de un cliente
Equipo de datos	Restauración de analítica sin enmascarar y dumps antiguos	Mal uso, compromiso	Confidencialidad de todas las columnas Restricted y Confidential	Alta	capítulos 12, 13	Los datos enmascarados aún pueden revelar valores por inferencia
Proveedor de contabilidad	Archivos exportados	Error, mal uso	Confidencialidad de las columnas exportadas y su tiempo de vida	Media	capítulos 12, 13	El proveedor conserva el archivo tras la entrega

Actor	Ruta	Clase principal	Impacto	Exposición	Se controla en	Riesgo residual
Pipeline de despliegue	Login admin guardado como variable del pipeline	Error, compromiso	Integridad del esquema y de los datos; todas las tablas	Media	capítulos 4, 11	Una migración revisada aún se ejecuta con los derechos del propietario
Personal de plataforma y DBA	Login admin compartido ahora; <code>ledgerly_access_admin</code> en el diseño objetivo	Error, mal uso, compromiso	Se pueden cambiar los grants, los controles y la evidencia	Media	capítulos 3, 7, 14	El rol existe, así que su mal uso es posible; solo queda la detección
Plano de control (activo)	Gateway, credential broker, almacén de auditoría	Error, compromiso	Todas las demás filas cambian de significado	Media	capítulos 4, 7, 14	El compromiso de un componente debilita la evidencia que guarda
Cualquier sesión humana	Una sentencia escrita a mano dentro de un rol concedido	Error	Sentencia sin límite, confirmada de inmediato	Alta	capítulos 9, 10	Una sentencia puede ser válida, estar aprobada y aun así ser errónea en su efecto
Cualquier origen de red	La VPN de la empresa llega directamente al primary	Compromiso	Una credencial filtrada funciona desde cualquier laptop de la VPN	Media	Capítulo 7	Una credencial robada desde un origen permitido sigue funcionando allí
Atacante externo	Cualquier credencial filtrada de las filas anteriores	Compromiso	Hereda el radio de impacto del propietario de la credencial	Baja a media	capítulos 4, 7, 14	Una credencial robada dentro de su período de vida funciona hasta que vence
Todos los actores	Ningún login con nombre en el log de la base de datos	Todas las clases	Rendición de cuentas de todas las demás filas	Alta	capítulos 7, 14	El registro es tan confiable como el almacén que lo guarda

De la tabla se desprenden cuatro observaciones, y son la razón para construirla.

Primero, el login admin compartido aparece en la mayoría de las filas. La tabla muestra un solo problema con muchas caras, y el primer control, un rol aparte para cada principal (Capítulo 3), atiende de golpe una gran parte de él. Un modelo que revela el orden en el que conviene trabajar está cumpliendo su función.

Segundo, entre las mayores exposiciones hay puertas que nunca se pensaron como puertas: la restauración de analítica y los dumps antiguos. Ninguna de ellas es una ruta de acceso en el diagrama mental que alguien tiene de la base de datos. Por eso los capítulos 6 y 13 tratan cada copia de datos como una puerta con un responsable y un vencimiento.

Tercero, la columna de riesgo residual es la parte honesta de la tabla. Todo control deja algo: un rol concedido aún puede usarse mal, una sentencia válida aún puede ser errónea, un registro solo es confiable hasta donde lo es su almacén. Una fila sin riesgo residual escrito es una fila en la que nadie preguntó.

Cuarto, las filas del plano de control, de la sesión humana y del origen de red nombran un activo o una ruta y no un actor, y la última fila tampoco nombra a un actor concreto. La falta de rendición de cuentas es una propiedad de todo el sistema, y eleva el costo de todas las demás filas, porque sin un login con nombre ningún incidente puede acotarse. El Capítulo 14 cierra esa fila. Los capítulos anteriores construyen las identidades que hacen posible cerrarla.

Advertencia: Un modelo de amenazas terminado es un mapa de dónde es más débil Ledgerly. Guárdalo en un lugar con las mismas restricciones de acceso que los sistemas que describe, y no lo pegues en un ticket público ni en un wiki de audiencia amplia. Un documento que enumera copias sin enmascarar y credenciales compartidas es útil para quienes lo mantienen y para nadie más.

Mantener vivo el modelo

Un modelo de amenazas escrito una sola vez es una instantánea, y la tabla anterior queda desactualizada el día en que Ledgerly añade una integración o el equipo de datos recibe una copia nueva. Lo trataría como un documento vivo con un responsable con nombre, como un runbook, guarda-

do en un repositorio con las mismas restricciones de lectura que los sistemas que describe, ya que la tabla enumera debilidades. Esto tiene un costo. La configuración de acceso suele vivir en un repositorio más ampliamente legible, de modo que un cambio allí no aparece automáticamente en la revisión del modelo. Aceptaría ese costo y cerraría la brecha con una regla: un pull request que cambia la configuración de acceso enlaza al cambio correspondiente del modelo, o explica por qué no hace falta ninguno.

Algunos sucesos deberían disparar una actualización sin que nadie tenga que acordarse:

- Un servicio, job, proveedor o herramienta nuevos que lleguen a los datos de producción.
- Una copia de datos nueva: una réplica nueva, un export, un refresco de staging, un dump para un análisis puntual.
- Un cambio en quién puede llegar a qué: una reorganización, una nueva rotación on-call, el fin de un contrato con un proveedor.
- Un incidente o un casi incidente, que es evidencia de qué fila estaba equivocada.
- Un hallazgo en una revisión de accesos (Capítulo 14), que es evidencia de qué suposición se desvió.

Cada fila debería registrar lo suficiente para que un extraño pueda cuestionarla: el actor, la ruta, la clase, el impacto, el juicio de exposición, el control y su responsable, la fecha de la última revisión y la suposición de la que depende la fila. El último campo es el que los equipos se saltan y el más útil: “el export del proveedor contiene solo estas columnas” es una afirmación que se puede comprobar, y una fila que depende de una afirmación sin comprobar es una conjetura.

Una pequeña comprobación puede encontrar las filas que han quedado obsoletas. No decide nada. Señala a una persona las filas con más probabilidad de estar equivocadas, y trata una fecha de revisión ausente como obsolescencia. Además toma como entrada los sucesos enumerados arriba: cuando un pipeline crea un login, un rol o una copia de datos, registra un evento, y un evento que ninguna fila cubre abre un elemento de revisión, de modo que la actualización no depende de que alguien se acuerde.

```

function findStaleThreatRows(rows, maxAgeDays, accessEvents):
    stale = []
    // an event from the pipeline: a new login, role or data copy
    for each event in accessEvents:
        if not anyRowCovers(rows, event):
            stale.append(newReviewItem(event))
    for each row in rows:
        // a row with no review date has never been reviewed
        if row.last_reviewed is null:
            stale.append(row)
        else if now() - row.last_reviewed > maxAgeDays:
            stale.append(row)
        // a control with no owner is a control nobody maintains
        else if row.control_owner is null:
            stale.append(row)
    if count(stale) > 0:
        notify("platform", "threat model rows need review")
    return stale

```

La cadencia es un trade-off. Una pasada trimestral es algo que un equipo pequeño puede sostener, y una anual rara vez basta, pero yo elegiría el intervalo según la rapidez con que cambien las puertas de Ledgerly. Los contactos de seguridad y de cumplimiento fijan las preguntas que plantea la revisión, y el equipo de plataforma las responde con hechos sobre el sistema.

Trade-offs y costos

Un modelo de amenazas cuesta tiempo escribirlo y más tiempo mantenerlo. Para un equipo pequeño, la primera versión plausiblemente requiere unos días de discusión cuidadosa, y el mantenimiento es una tarea recurrente que compite con el trabajo de funcionalidades. Si nadie es responsable de él, el costo se paga una vez y el beneficio se desvanece. En un sistema muy pequeño, con un servicio y una base de datos, la tabla completa es más de lo que el problema necesita; basta media página que enumere las puertas y el radio de impacto del peor caso de cada una.

El modelo también puede generar una falsa confianza: una tabla ordenada sugiere que todas las amenazas están en ella. Enumera las amenazas en las que alguien pensó, y una puerta que no está en la lista le resulta invisible. La mejor defensa son los disparadores de actualización de arriba, y el hábito de preguntar tras cada incidente: “¿qué fila habría predicho esto y, si ninguna, cuál es la fila que falta?”.

Los juicios de exposición son subjetivos. Dos ingenieros valorarán la misma puerta de forma distinta, y el desacuerdo es informativo, pero una valoración implica más precisión de la que respalda la evidencia. La mantendría en tres niveles y me resistiría a calcular puntuaciones.

Hay un costo humano en enumerar los grupos de personas como actores. Llamar actor de amenaza a la ingeniería on-call o al soporte puede leerse como una acusación. La respuesta está en el encuadre: el modelo describe rutas y roles, no individuos, y sus entradas dicen lo que el diseño permite. Los equipos de la tabla aparecen porque son los que tienen la necesidad más legítima de acceder a los datos.

Cuándo no usar el modelo completo: si nadie puede responder hoy a “¿quién tiene acceso?”, empieza por un inventario de logins y de copias de datos antes de escribir amenazas, porque un modelo construido sobre un inventario inexacto describe un sistema que no existe. Y no lo uses como sustituto de una decisión. Ordena y explica; el equipo sigue eligiendo qué controles construir primero y qué riesgos aceptar por escrito.

Errores comunes

- **Modelar solo al atacante externo.** Coincide con la imagen habitual de la seguridad, y los controles son conocidos. El costo es que los errores y la curiosidad quedan sin atender. Da a cada una de las tres clases sus propias filas en la tabla.
- **Enumerar controles en lugar de amenazas.** “Tenemos una VPN” registra una respuesta sin la pregunta. Indica primero lo que puede salir mal y después el control, para poder retirar un control cuando su amenaza ya no se aplique.
- **Dejar fuera a los actores que no son personas.** Los pipelines, los jobs y los exports se pasan por alto con facilidad porque nadie inicia sesión como ellos, y suelen tener el mayor poder permanente. Enumera cada principal y cada export.
- **Tratar los datos como uniformemente sensibles.** Si todas las columnas son “críticas”, el modelo no da ninguna guía sobre por dónde empezar. Usa las cuatro clases de datos (Restricted, Confidential, Internal, Public) para que el peso de una amenaza siga el peso de los datos.

- **Saltarse la disponibilidad.** Una consulta descontrolada no parece un suceso de seguridad, así que se escapa de las revisiones de seguridad. Para un negocio de pagos puede importar tanto como uno. Dale a la disponibilidad sus propias filas.
- **Escribirlo una sola vez.** El modelo es exacto durante unas semanas y después se vuelve folclore. Asigna un responsable y añade disparadores de actualización.

Ideas clave

- Un modelo de amenazas es una lista estructurada y breve de actores, rutas, clases de amenaza, impactos y controles, y su trabajo es permitir que un equipo compare las puertas entre sí.
- Los errores, el mal uso y el compromiso necesitan controles distintos, y un diseño que solo considera a los externos deja sin atender las dos primeras clases; aprobar la intención atiende el mal uso, mientras que aprobar la sentencia y su efecto esperado atiende los errores.
- Todo actor pertenece a la lista, incluidos los pipelines, los jobs por lotes, los exports a proveedores y el personal de plataforma que administra los controles, porque los principales en cuyo nombre nadie inicia sesión suelen tener el mayor poder permanente, y el plano de control es también un activo.
- La confidencialidad, la integridad y la disponibilidad necesitan cada una sus propias filas, y la rendición de cuentas, la capacidad de decir qué principal hizo qué, eleva el costo de todas las demás amenazas cuando falta.
- El radio de impacto siempre se refiere a un login, una credencial o una acción concretos, y se describe en siete dimensiones: datos, acciones, entornos, tiempo, copias, sentencias y origen de red.
- En el estado inicial de Ledgerly un único login admin compartido aparece en casi todas las filas, así que un rol aparte para cada principal es el control que reduce de golpe la mayor parte de la tabla.
- El modelo vale lo que valga su mantenimiento: dale un responsable, ata las actualizaciones a sucesos como los servicios nuevos y las copias de datos nuevas, y guárdalo donde solo puedan leerlo quienes mantienen el sistema.

El modelo te dice qué puertas importan más, y en el caso de Ledgerly casi todas llevan de vuelta a un solo login. El Capítulo 3 empieza a desmontar ese login: quiénes son los principales, qué debe contener un rol y cuán poco necesita cada uno.

Capítulo 3 — Identidades, roles y mínimo privilegio

Imagina un jueves por la tarde en Ledgerly. Una tabla nueva, `invoice_disputes`, entró en producción el martes, y el contacto de seguridad le hace al equipo de plataforma una pregunta breve y razonable: ¿quién puede leerla? Alex empieza por los grants y no encuentra ninguno, porque todos se conectan como `admin`, que es propietario de la tabla y puede leerlo todo. Entonces Alex busca quién la leyó de verdad. El log de la base de datos tiene miles de líneas sobre la tabla, todas escritas por el mismo login: el worker, el informe nocturno, dos ingenieros en sus laptops y la herramienta de soporte.

Nadie puede ubicar la tabla nueva. La respuesta honesta a “¿quién puede leerla?” es “quien pueda llegar a la base de datos”, y la respuesta honesta a “¿quién la leyó?” es “alguien”. Nadie hizo nada mal. El diseño nunca le pidió a la base de datos que distinguiera entre quienes llaman, ni que limitara a ninguno.

El Capítulo 1 dividió el acceso en cinco partes: **Identidad**, **Grant**, **Ruta**, **Ejecución** y **Registro**. La tesis del libro es que el acceso es un problema de autorización, ejecución y rendición de cuentas, no de distribución de credenciales. Este capítulo sirve al pilar de la autorización. Cubre las dos primeras partes, quién pregunta (Identidad) y qué puede hacer ese principal (Grant), y señala dónde terminan los grants, porque ahí continúa la parte Ejecución (Capítulo 9).

Añade los primeros pasos del estado objetivo del ejemplo continuo del libro. Todas las tablas pertenecen a `ledgerly_owner`, y nadie se conecta como el propietario. Cada servicio recibe su propio rol de aplicación con los grants más estrechos que funcionen. El capítulo introduce además el esquema completo de roles, la manera en que los principales llegan a los roles, los grants a nivel de columna y el techo del acceso just-in-time. Los capítulos posteriores se apoyan en estas definiciones y no las repiten.

El problema y por qué es difícil

Una base de datos responde dos preguntas en cada sentencia: quién es este y si ese principal tiene permitido hacer esto. Una cuenta compartida vuelve inútil la primera respuesta, y una cuenta amplia hace que la segunda sea siempre un sí. Ambas son fáciles de alcanzar. Un login, una contraseña y todos los derechos es lo más barato que funciona el día en que un equipo crea su primera base de datos.

El costo llega después, en forma de evidencia ausente. Cuando las personas y los servicios que comparten el login admin (digamos, veinte personas, cinco servicios, el pipeline de despliegue y el paso de export) lo usan todos, el log de la base de datos no puede decir cuál de ellos ejecutó una sentencia. El log de la aplicación describe lo que la aplicación pretendía hacer, no lo que llegó a la base de datos. Eso es un problema de diseño, no de disciplina, porque ningún cuidado individual devuelve la información que el diseño descartó.

El remedio obvio es escribir una política: concede solo lo necesario. Una política no funciona aquí, porque la base de datos hace que la opción amplia sea la fácil. Conceder todo requiere una sentencia y nunca falla. Conceder el conjunto más estrecho que funcione requiere conocer cada consulta que ejecuta un workload, incluido el job mensual que nadie recuerda y la ruta de retry que se ejecuta dos veces al año. Un grant demasiado estrecho rompe producción en el peor momento. Un grant demasiado amplio no rompe nada, así que nadie lo nota. Con esa asimetría, los privilegios solo crecen.

La propiedad añade una segunda trampa. En la mayoría de los motores, el propietario de una tabla puede cambiar su estructura, eliminarla y conceder acceso a otros. Si un servicio se conecta como propietario porque crea tablas al arrancar, el servicio tiene el poder de reescribir el modelo de acceso de la base de datos. Si el propietario es una persona, el modelo depende del empleo de esa persona. La propiedad es un privilegio estructural, y no debería residir en nada que atienda tráfico ni en nada que atienda una carrera profesional.

El remedio es una estructura, no una instrucción: un login distinto por principal, un pequeño conjunto de roles que llevan los privilegios y una manera de mantener ambos alineados con lo que el equipo pretende.

Principales, logins y roles

Este capítulo trabaja sobre Identidad y Grant, y toca Ruta donde un rol dice desde dónde puede conectarse. Un **principal** es cualquier cosa que se autentica. En el mundo de Ledgerly hay cuatro tipos que conviene separar.

- **Personas.** Ingenieros de backend on-call, personal de soporte, el equipo de datos y el equipo de plataforma. Llegan y se van, cambian de equipo y trabajan desde sitios distintos.
- **Servicios.** `billing-api`, `invoice-worker` y `payments-service`. Son de larga duración, se ejecutan en muchas instancias y soportan el tráfico más pesado.
- **Jobs.** `reporting-job` y su paso de export. Se ejecutan según un calendario y terminan. Un job suele necesitar menos que un servicio, y durante menos tiempo.
- **Herramientas.** `admin-console` es el caso más claro. Es un servicio por cómo se conecta, pero actúa en nombre de una persona, así que en cada acción intervienen dos principales: la herramienta y la persona que está detrás.

El proveedor externo de contabilidad nunca se conecta a ninguna base de datos de Ledgerly; recibe archivos, y el Capítulo 13 trata esa ruta.

Identidad, login y rol son tres cosas distintas

El libro mantiene separadas tres palabras, porque mezclarlas es el origen de la mayor parte de la confusión en el diseño de accesos.

Una **identidad** es la afirmación verificada de que un principal es quien dice ser. Un **login** es la cuenta, a nivel de base de datos, con la que se conecta un principal. Un **rol** es un conjunto de privilegios con nombre. Un principal tiene una identidad, se conecta mediante un login, y el login es miembro de uno o más roles. El login es lo que aparece en el log de la base de datos. El rol es lo que decide qué puede hacer el login.

Esto deja al modelo una frase que vale la pena recordar: nadie inicia sesión como un rol. Un rol existe para concederse a logins y para contener privilegios, y nunca es una persona. Si el log muestra el nombre de un rol donde debería aparecer una persona o una instancia de workload, el modelo se ha eludido en algún punto.

Para las personas, el access gateway (Capítulo 7) crea un login por sesión que lleva la identidad de la persona, y ese login recibe `human_readonly` por defecto. Para los servicios, cada instancia se conecta con un login que es miembro de su rol `app_*`. En el estado objetivo ese login es dinámico y de vida corta (Capítulo 4). En el estado inicial es el login admin compartido. La consecuencia para la pregunta central es clara: el log nombra un login, y un login corresponde a una persona o a una instancia de workload, de modo que “¿quién ejecutó esa sentencia?” tiene respuesta.

Por qué una cuenta compartida destruye la evidencia

El Capítulo 1 expuso el argumento completo; la versión corta es que una cuenta compartida no puede atribuir una sentencia, no puede revocarse para una sola persona y no puede limitarse por debajo de lo que necesita quien llama con el alcance más amplio.

Hay una ruta que requiere cuidado: `app_admin_console` es un workload, pero sus usuarios son personas. La consola pasa la identidad del principal humano con cada acción, y el esquema siguiente muestra cómo.

```
function consoleAction(person, action):
  correlationId = newCorrelationId()
  outcome = "failed"
  // set on the session the console borrowed, cleared when it is
  returned
  setSessionContext({ actor: person.id, correlation: correlationId
})
  try:
    run(action)
    outcome = "done"
  finally:
    clearSessionContext()
    // a failed or half-finished action is recorded too
    auditLogAppend({ actor: person.id, correlation: correlationId,
                    workload: "admin-console", action:
action.name,
                    result: outcome })
```

El login de la base de datos nombra al workload, y el contexto de sesión y `audit_events` llevan a la persona y el correlation id, de modo que el log de la aplicación, la traza de auditoría y el propio registro de la sentencia comparten una misma clave. El Capítulo 14 cubre cómo se unen los regis-

tros. El contexto lo fija la consola, por lo que es una afirmación hecha por el workload y no un hecho verificado por la base de datos; la salvedad sobre el pooler de la sección de reglas de fila se aplica también a él.

Una jerarquía de roles con cuatro capas

Construiría el esquema de roles en cuatro capas, desde la que nadie usa hacia afuera, hasta las que usan las personas. Cada capa tiene un solo trabajo.

Capa uno: el propietario

`ledgerly_owner` es propietario de todos los objetos del esquema: esquemas, tablas, vistas y secuencias. Nadie inicia sesión con él. Existe para que la propiedad no esté ligada ni a una persona ni a un servicio.

```
CREATE ROLE ledgerly_owner;  
// no login is ever created for this role; objects are created  
// under it only by a migration that assumes it  
  
CREATE ROLE ledgerly_migrator;  
GRANT ledgerly_owner TO ledgerly_migrator;  
// membership grants the owner's privileges, not ownership: objects  
// belong to the active role, so the migration always assumes  
// ledgerly_owner explicitly, on every engine
```

Capa dos: el migrator

`ledgerly_migrator` ejecuta los cambios de esquema. El pipeline de despliegue usa su login, las personas no, y no tiene privilegios de datos propios. Es miembro de `ledgerly_owner`. En algunos motores esa pertenencia transmite los privilegios del propietario sin ningún paso explícito, pero no la propiedad: un objeto nuevo pertenece al rol que esté activo cuando se crea, así que la migración siempre asume `ledgerly_owner` de forma explícita.

El migrator tiene privilegios permanentes de propietario, ejercidos mediante una credencial de vida corta. Los privilegios nunca vencen; solo vence cada credencial. Tres cosas lo acotan: la credencial vive solo en el pipeline, es de vida corta (capítulos 4 y 11) y el procedimiento del pipeline es la única forma autorizada de usarla. Los cambios de esquema son posibles sin que ninguna persona ni ningún servicio tenga la propiedad, y

si esos límites se sostienen, cada uso del propietario corresponde a una ejecución del pipeline. El Capítulo 11 cubre cómo se revisan las migraciones.

Los backfills que forman parte de un cambio de esquema se ejecutan dentro de la migración como `ledgerly_migrator`. Cualquier otro cambio masivo de datos es una corrección de datos y sigue el Capítulo 10. Trazo esa línea porque el poder del migrator es grande y no debería convertirse en la ruta para todo cambio incómodo.

Capa tres: roles de aplicación

Cada workload recibe su propio rol. Un rol por servicio cuesta cinco definiciones que mantener alineadas con cinco bases de código. A cambio, el radio de impacto de cada credencial es del tamaño del trabajo de ese servicio: si se filtra la de `invoice-worker`, expone lo que el worker podía hacer, y el worker nunca necesitó `payment_tokens`.

Los roles son `app_billing_api`, `app_invoice_worker`, `app_payments_service`, `app_reporting_job` y `app_admin_console`. El prefijo dice quién lo usa: `app_` para los workloads, `human_` para las personas. Los roles no llevan nombres de personas ni el nombre de un entorno.

Capa cuatro: roles humanos

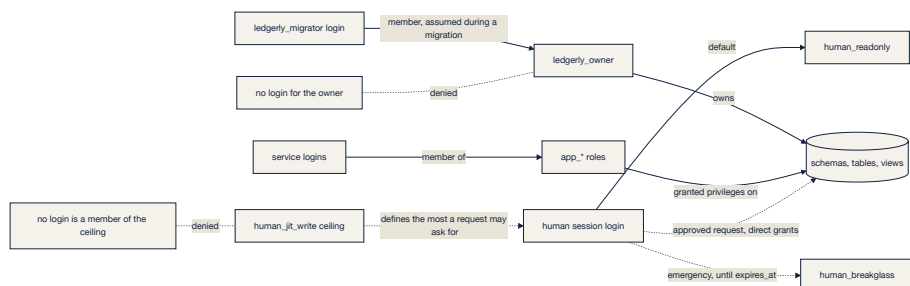
Las personas reciben tres roles, y se diferencian en cómo se mantienen.

- `human_readonly` es el valor por defecto. Lee para investigar, retiene las columnas Restricted y enmascara las columnas Confidential donde lo indique el Capítulo 12.
- `human_jit_write` es un techo. Nunca se concede a un login. Cada solicitud aprobada concede un subconjunto de él, sobre tablas y columnas con nombre, directamente al login de la sesión, y luego lo retira.
- `human_breakglass` es un acceso amplio de emergencia con registro intensivo (el Capítulo 8 lo define) y revisión obligatoria posterior.

El techo es el menos intuitivo de los tres, y vuelvo a él después de explicar la mecánica de los grants.

El diagrama

El diagrama siguiente muestra quién es miembro de qué y quién es propietario de qué. Fíjate en las dos cosas que no están conectadas: no existe ningún login para `ledgerly_owner`, y ningún login es miembro de `human_jit_write`.



Grants: esquema, tabla y columna

Un **grant** es el acto, y el registro resultante, de dar un privilegio a un rol, dar un rol a un login o a otro rol o, solo para el acceso JIT, dar un privilegio directamente a un login de sesión. Un **privilegio** es lo específico que permite un grant, como `SELECT` sobre una tabla. Los tres significados mantienen preciso el vocabulario, y en este capítulo diré a cuál me refiero.

Empezar desde cero

El punto de partida de cualquier rol es no tener ningún privilegio. Añade grants hasta que el workload funcione, y detente. Los motores difieren en lo que puede hacer por defecto un login nuevo: en algunos, cualquier login puede ver los nombres de los objetos o crear objetos en un esquema por defecto. Enumera lo que recibe automáticamente cada login y elimina lo que un rol no necesita.

Los privilegios se aplican en tres niveles, y se suman: un rol tiene la unión de todo lo que se le ha concedido en cualquier nivel, y un grant estrecho nunca anula uno amplio.

- **Nivel de esquema.** Si un rol puede ver siquiera los objetos de un esquema. En los motores que lo exigen, los grants a nivel de tabla no sirven de nada sin él, lo que convierte el acceso al esquema en un interruptor grueso útil.

- **Nivel de tabla.** `SELECT`, `INSERT`, `UPDATE` y `DELETE` sobre una tabla completa.
- **Nivel de columna.** Los mismos privilegios, solo sobre columnas con nombre.

Los grants a nivel de columna importan porque los datos sensibles están repartidos de forma desigual: `customers` contiene datos de contacto Confidential, `bank_accounts` contiene `iban` y `account_holder_name`, que son Restricted, y `payment_tokens` contiene `token_value`, que es Restricted. Un grant a nivel de tabla dice que sí a todas las columnas de la tabla. Como los privilegios se suman, restringir columnas significa conceder solo a nivel de columna y no tener ningún grant a nivel de tabla sobre esa tabla para el rol, ni directamente ni a través de otro rol al que pertenezca.

Grants de las aplicaciones

Los dos primeros roles de aplicación, a nivel de tabla:

```
GRANT SELECT, INSERT, UPDATE ON customers, invoices, invoice_lines TO
app_billing_api;
GRANT SELECT ON payments, ledger_entries TO app_billing_api;
GRANT INSERT ON audit_events TO app_billing_api;

GRANT SELECT, INSERT, UPDATE ON invoices, invoice_lines TO
app_invoice_worker;
GRANT SELECT ON customers TO app_invoice_worker;
GRANT INSERT ON audit_events TO app_invoice_worker;

// not granted to either role: bank_accounts, payment_tokens,
// UPDATE or DELETE on ledger_entries and audit_events
```

`app_billing_api` lee `ledger_entries` y `payments` pero no puede escribirlas, porque solo `payments-service` escribe `payments`, `bank_accounts` y `payment_tokens` e inserta `ledger_entries`. Ninguno de los dos roles tiene `DELETE` sobre nada; si un servicio lo necesita, el grant se añade para esa tabla con una razón, y la revisión lo nota.

Grants a nivel de columna para los otros roles

Los roles con una daga en la matriz están restringidos por columnas. Las listas de columnas siguientes son ilustrativas: qué columnas Confidential aparecen, y en qué forma enmascarada, lo decide el Capítulo 12.

```
// illustrative column lists
GRANT SELECT (id, created_at, status) ON customers TO
app_reporting_job;
GRANT SELECT (id, customer_id, amount_minor, currency, status)
    ON payments TO app_reporting_job;
GRANT SELECT ON invoices, ledger_entries TO app_reporting_job;

GRANT SELECT (id, created_at, status) ON customers TO
app_admin_console;
GRANT SELECT, UPDATE (status) ON invoices TO app_admin_console;
// UPDATE (status) is a column-level grant: the tool can change an
invoice's
// status and no other column of the row
```

El último grant es el que más me gusta. `app_admin_console` puede cambiar el `status` de una factura y nada más de la fila, de modo que el grant declara la intención y la base de datos la hace cumplir aunque el código de la herramienta cambie más adelante. `app_reporting_job` no tiene ningún privilegio de escritura sobre ninguna tabla principal: escribe sus resultados en `ledgerly_analytics`.

Nota: En los motores que replican de forma física, los roles se definen una sola vez para todo el clúster, así que desde dónde puede conectarse cada rol es una regla de la capa de conexiones (capítulos 5 y 7) y no algo que expresen los grants. El Capítulo 6 da el detalle.

La matriz como referencia

La tabla siguiente es la referencia del estado objetivo para los seis roles que leen o escriben las tablas principales. S es `SELECT`, I es `INSERT`, U es `UPDATE`, D es `DELETE`, un guion significa sin acceso y una daga (†) significa restringido por columnas: se retienen las columnas `Restricted` y se enmascaran las columnas `Confidential` donde lo indique el Capítulo 12.

Tabla	app_ billing_ api	app_ invoice_ worker	app_ payments_ service	app_ reporting_ job	app_admin_ console	human_ readonly
customers	S, I, U	S	S	S†	S†	S†
bank_ accounts	—	—	S, I, U	—	—	—
payment_ tokens	—	—	S, I, U, D	—	—	S†
payments	S	—	S, I, U	S†	S†	S†

Tabla	app_ billing_ api	app_ invoice_ worker	app_ payments_ service	app_ reporting_ job	app_admin_ console	human_ readonly
invoices	S, I, U	S, I, U	S	S	S, U (solo status)	S
invoice_ lines	S, I, U	S, I, U	—	S†	S†	S†
ledger_ entries	S	—	S, I	S	—	S
audit_ events	I	I	I	—	I	S

Leo la matriz por columnas. Cada columna es la descripción del puesto de un workload escrita como privilegios, y quien revisa puede preguntar si sigue siendo cierta. A lo largo de ella, solo `payments-service` alcanza `bank_accounts`, solo `payments-service` toca `payment_tokens`, aparte de una lectura restringida por columnas de `human_readonly`, y ningún rol humano escribe por defecto.

Append-only es estructural

Ningún rol de aplicación, ningún rol humano y ningún grant JIT tiene `UPDATE` ni `DELETE` sobre `ledger_entries` ni `audit_events`, y eso incluye a `human_breakglass`. Una corrección es una entrada nueva, nunca una edición. Las correcciones del ledger solo se emiten a través de la ruta de corrección de `payments-service`, nunca por un login humano.

Hay una excepción estructural: `ledgerly_owner`, y por tanto una migración en curso, puede alterar estas tablas, porque el propietario puede alterar o eliminar la tabla y concederse a sí mismo cualquier privilegio sobre ella. Por eso el Capítulo 11 revisa toda migración que las toque. El control está en la revisión, no en el privilegio.

Valores por defecto, propiedad y el superusuario

Privilegios por defecto

Muchos motores pueden hacer que un grant se aplique a las tablas creadas más adelante, ya sea como un privilegio por defecto asociado a un propietario o como un grant que abarca todo el esquema. Eso es cómodo al prin-

cipio y una trampa después, porque una tabla nueva hereda en silencio lo que diga el grant.

Mi preferencia es un valor por defecto vacío. Una tabla nueva no la lee nadie salvo el propietario, y la migración que la crea declara cada grant en el mismo archivo. El costo son unas pocas líneas más por migración; el beneficio es que una sola revisión muestra todo el efecto de acceso de un cambio.

Advertencia: Un privilegio por defecto que concede `SELECT` sobre tablas futuras a un rol amplio como `human_readonly` expone la siguiente tabla sensible en el momento en que se crea. Revisa primero esos valores por defecto y elimina cualquiera que cubra tablas futuras.

La notación es ilustrativa, porque los privilegios por defecto no tienen una sintaxis neutra.

```
// illustrative notation: defaults for objects created by
ledgerly_owner
DEFAULT PRIVILEGES FOR ledgerly_owner ON TABLES: none

// each migration that creates a table states its own grants
CREATE TABLE <new_table> (...);
GRANT SELECT ON <new_table> TO app_billing_api;
```

El peligro de la propiedad

En los motores comunes, el propietario puede eliminar un objeto, cambiar su definición, conceder privilegios a cualquiera y, en muchos casos, ignorar las restricciones a nivel de fila que se aplican a los demás. Quien tiene la propiedad puede cambiar el propio modelo de acceso.

Por eso las reglas en torno a `ledgerly_owner` son estrictas, y la rigidez es barata. No existe ningún login para él. Los roles de aplicación nunca son propietarios de tablas, ni siquiera de las que crean para su propio uso, porque “el servicio crea su tabla al arrancar” es la manera en que un workload acaba siendo propietario de parte del esquema. El propietario es el rol menos usado del sistema, que es el estado correcto.

El superusuario

La mayoría de los motores tienen un rol administrativo integrado que se salta las comprobaciones de privilegios, aunque los servicios gestionados pueden retenerlo. Aquí lo llamo el superusuario; queda fuera del esquema de roles de Ledgerly porque viene con el motor.

Lo trataría como un instrumento sellado. Ninguna aplicación, job ni sesión humana lo usa en el trabajo diario. Su credencial se guarda donde el equipo de plataforma pueda recuperarla y la recuperación deje un registro, y sirve para operaciones a nivel de motor que ningún rol puede hacer, como reparar la replicación. El límite honesto es que puede deshacer todos los controles de este capítulo, y no conozco ningún diseño que elimine ese poder sin hacer imposible la recuperación. Haz que su uso sea raro, visible y revisado; el Capítulo 15 vuelve sobre él. La administración rutinaria, como aplicar definiciones de roles o emitir logins, usa en su lugar `ledgerly_access_admin`. Si se usa el superusuario para tareas rutinarias, a algún rol le falta un privilegio que debería tener.

`ledgerly_access_admin` es un rol usado por tres principales, cada uno mediante su propio login: el paso del pipeline que aplica la definición de acceso, el credential broker (Capítulo 4) y el access gateway (Capítulo 7). Crea logins y gestiona membresías, no es propietario de ninguna tabla y no tiene privilegios de datos, y aun así es un poder amplio: quien lo tiene puede conceder cualquier cosa dentro de los roles que administra. Los logins separados mantienen a los tres distinguidos en el log, aunque no en lo que pueden hacer.

Lo que acota el poder del propio plano de control no es por tanto el rol sino el entorno. Cada uno de los tres logins solo es accesible desde su propio componente. Cada acción va al registro de auditoría (Capítulo 14), y la comprobación de drift que aparece más abajo compara el resultado con el código. Los cambios en la definición de acceso pasan por su propio conjunto reducido de revisores. Estos límites hacen visible el mal uso; no lo hacen imposible, y yo lo diría así en la revisión de accesos.

Estrechar más: vistas, reglas de fila y tenants

Vistas

Una **vista** es una consulta almacenada que se comporta como una tabla. Un rol con acceso a la vista lee lo que expone la consulta sin tener un privilegio sobre la tabla subyacente.

```
// illustrative: a narrowed projection owned by ledgerly_owner
CREATE VIEW customer_support_summary AS
    SELECT id, created_at, status FROM customers;
GRANT SELECT ON customer_support_summary TO app_admin_console;
```

Nota: En muchos motores una vista se ejecuta con los privilegios de su propietario, no con los de quien la llama. Eso es lo que hace que funcione el estrechamiento, y es también la razón por la que una vista que selecciona demasiado lo filtra a todos los roles que tienen un grant sobre ella. Algunos motores permiten además elegir el otro comportamiento. Comprueba qué hace el tuyo por defecto y prueba la vista con un rol que no tenga privilegios sobre la tabla base.

Prefiero los grants a nivel de columna donde el motor los admite, porque el grant vive junto a la tabla. Una vista es mejor cuando lo importante es la forma: un join, un valor calculado o una interfaz estable sobre una tabla cambiante. El libro no usa una vista por solicitud para el acceso JIT, por dos razones: una vista debe tener propietario, así que el gateway que la crea necesitaría poder de propietario, y una vista creada para cada solicitud rompe el modelo de propiedad.

Reglas de fila y tenants: una regla de decisión

Una regla de fila estrecha las filas dentro de una celda de la matriz y no cambia lo que dice la celda. Soporte mira a un cliente a la vez, así que una regla como “app_admin_console solo ve las filas de `invoices` donde `customer_id` es igual al cliente del caso” encaja donde un filtro de columnas no. Los motores discrepan en la sintaxis, por lo que el libro no muestra ninguna notación.

Los clientes de Ledgerly son empresas, y cada una espera no ver las facturas de otra. Ese aislamiento es sobre todo trabajo de la aplicación:

`billing-api` filtra por cliente en cada consulta, y ningún rol separa a un tenant de otro, porque el rol de la aplicación debe leerlos todos para atender a cualquiera.

Usa una regla de fila, para soporte o como segunda capa de tenants, solo cuando se cumplan dos condiciones: una fila equivocada es grave y la aplicación fija el valor de contexto de manera confiable en cada sesión. El costo es que cada sentencia lleva el predicado y algo debe fijar el contexto en cada sesión. Detrás de un pooler, un valor fijado en una sesión puede llegar a la siguiente persona que tome prestada la misma conexión (el Capítulo 5 vuelve sobre esto). Cuando falta el contexto, la regla debe devolver cero filas, y la primera prueba es ese caso. Una regla que falla en abierto es peor que ninguna. Las bases de datos separadas son una frontera más fuerte a un precio operativo mucho mayor; cuál conviene depende de cuántos tenants tengas y de lo que les hayas prometido.

El techo para el acceso just-in-time

`human_jit_write` define el techo: lo máximo que puede pedir una solicitud just-in-time (JIT), expresado como privilegios sobre tablas y columnas con nombre. El acceso JIT es un acceso concedido para una tarea concreta y por un tiempo limitado; el Capítulo 8 cubre las aprobaciones y las duraciones, y este capítulo solo define el techo.

El rol nunca se concede a un login. Una solicitud es una lista de tripletas (privilegio, tabla, columnas) que cabe dentro del techo, guardada con la solicitud. Al aprobarse, exactamente esas tripletas se conceden directamente al login de sesión del solicitante, y al vencer, exactamente esas tripletas se revocan. Un rol compartido no puede limitarse por solicitud, así que el alcance de cada solicitud vive en los grants y el rol traza el borde exterior.

```

// the ceiling: what a JIT request may ask for, never granted to a
login
GRANT SELECT ON customers, payments, invoices, invoice_lines
    TO human_jit_write;
// id and created_at are assigned by the database, never changed or
supplied
// by a data fix, so the write privileges below name every other
column
GRANT UPDATE (<every column except id, created_at>)
    ON customers, payments, invoices, invoice_lines TO
human_jit_write;
GRANT INSERT (<every column except id, created_at>)
    ON customers, payments, invoices, invoice_lines TO
human_jit_write;
// DELETE only where a request names it, and never on a table with
// Restricted columns
GRANT DELETE ON customers, payments, invoices, invoice_lines
    TO human_jit_write;

// illustrative column lists; Restricted columns are not named
anywhere
GRANT SELECT (customer_id), UPDATE (customer_id)
    ON bank_accounts TO human_jit_write;
GRANT SELECT (customer_id, provider, last_four, expires_on),
    UPDATE (customer_id, provider, last_four, expires_on)
    ON payment_tokens TO human_jit_write;
// no table-level SELECT on either table: it would include the
Restricted columns
// no INSERT or DELETE on either: a new or removed row sets or
destroys
// Restricted values

```

Lee el bloque por lo que deja fuera. `bank_accounts` y `payment_tokens` están abiertas solo para las pocas columnas con nombre que no son Restricted, y nunca para identificadores de fila ni marcas de tiempo: `iban`, `account_holder_name` y `token_value` nunca pueden aparecer en una solicitud JIT. `INSERT` se limita a las cuatro primeras tablas, y no puede nombrar `id` ni `created_at`, de modo que una fila restaurada o nueva siempre recibe valores asignados por la base de datos. Las dos tablas append-only y los registros de solicitudes de acceso, que pertenecen al gateway, quedan por completo fuera del techo. Toda tripleta, salvo un `DELETE`, nombra sus columnas.

Aquí se declaran dos costos, una sola vez. Las columnas que vinculan una fila con su propietario, como `customer_id`, no son Restricted, así que un `UPDATE` JIT puede cambiarlas. Un cambio de vínculo con el propietario de este tipo sigue dentro del techo, pero es una clase de operación peligrosa:

la política de ejecución lo somete al gate y se encamina a la ruta de producto (Capítulo 10) en lugar de ejecutarse como SQL directo. El control son el gate y la ruta de producto, no solo la aprobación four-eyes. Y un SELECT JIT muestra sin enmascarar cualquier columna Confidential que nombre, lo cual forma parte de lo que acepta quien aprueba.

Lo que mantiene honesto al techo es una comprobación, no la confianza. El gateway rechaza cualquier solicitud fuera de él (el Capítulo 8 muestra la comprobación), y la prueba de drift que sigue afirma que ningún login es miembro de `human_jit_write`, el fallo que convertiría un techo en un grant permanente.

Donde el motor no tiene grants a nivel de columna

No todos los motores admiten grants a nivel de columna. El libro usa exactamente una alternativa: donde el motor no puede conceder sobre columnas, las columnas Restricted se trasladan a tablas aparte, `bank_account_secrets` y `payment_token_values`.

```
table bank_account_secrets:      // owned by ledgerly_owner
  bank_account_id    text primary key
  iban               text not null
  account_holder_name text not null

table payment_token_values:      // owned by ledgerly_owner
  payment_token_id    text primary key
  token_value         text not null

// reachable by app_payments_service and human_breakglass only;
// outside the ceiling, never part of a JIT request
```

Con la alternativa, las tablas principales no contienen ninguna columna Restricted, así que un grant JIT a nivel de tabla sobre ellas no puede alcanzar datos Restricted. El costo: el detalle de la solicitud baja al nivel de tabla, y quien aprueba ha aprobado todas las columnas no Restricted de la tabla, incluidas las Confidential, no solo las que nombró el solicitante.

Break-glass y las operaciones que están por encima del esquema

`human_breakglass` es el único rol humano que lee columnas Restricted. Lee todas las tablas y escribe con `INSERT` y `UPDATE` en todas salvo las dos `append-only`; en `bank_accounts` y `payment_tokens` solo escribe con `UPDATE`, y solo en columnas que no son Restricted. Es para emergencias, no para correcciones rutinarias, y no aparece en la matriz porque solo lo tiene una sesión, durante una ventana corta; ningún workload permanente lo tiene.

Las columnas Restricted solo se cambian a través de una ruta de aplicación. Una persona nunca las escribe con SQL directo, ni en una sesión JIT ni de otro modo: `admin-console` se lo pide a `payments-service`, el único que escribe, después de que una segunda persona apruebe, y el cambio queda registrado en `audit_events`. El Capítulo 10 describe esa ruta. Estos hechos marcan el borde de lo que harán alguna vez los grants. Un grant dice a qué tablas y columnas puede acceder una sesión; no puede decir qué sentencia, cuántas filas, ni si este cambio concreto es el que se aprobó. Eso es trabajo de la parte Ejecución, y el Capítulo 9 cubre cómo el gateway comprueba cada sentencia. Saber dónde se detienen los grants ayuda a decidir dónde encaja una capacidad nueva.

Roles como código y drift

Un esquema de roles que solo existe en la base de datos es uno que nadie puede revisar. Los grants de arriba son lo bastante cortos para leerse, y deberían residir en el control de versiones, cambiar mediante revisión y ser aplicados por el pipeline, igual que el esquema que protegen.

Dónde viven las definiciones

Mantendría dos tipos de archivo. Las migraciones crean las tablas y declaran los grants de cada tabla. Una definición de acceso aparte y revisada enumera los roles, las membresías entre roles y el techo. El pipeline aplica ambos como pasos separados. La definición de acceso se aplica mediante el login propio del paso del pipeline en `ledgerly_access_admin`, el mismo rol que usan el broker y el gateway mediante sus propios logins; no tiene privilegios de datos. Crear roles es un poder mayor que cambiar ta-

blas, así que su paso debería tener su propio conjunto reducido de revisores. La pregunta de quien revisa no es si el SQL es válido sino si este grant es el más estrecho con el que el cambio funciona. Una persona con la matriz abierta puede responderla en un minuto.

Estrechar un sistema existente

Ledgerly parte de un login compartido que es propietario de todo. No pasaría de ahí a la matriz en un solo cambio, porque un grant demasiado estrecho rompe producción y el equipo no puede saber qué se le pasó hasta que algo falla. Un camino de cuatro pasos:

1. **Dale primero a cada workload su propio login, con los mismos privilegios de antes.** No se rompe nada, y el log empieza de inmediato a nombrar al workload.
2. **Observa.** Usa las sentencias registradas bajo cada login para enumerar lo que el workload toca de verdad, durante un período lo bastante largo para incluir el job mensual lento. Un log de sentencias puede contener valores literales, así que trátalo como sensible: puede contener datos de clientes o valores Restricted que se pegaron en una consulta. Restringe quién lo lee, fija su retención y prefiere registrar las formas de las sentencias con parámetros donde el motor lo ofrezca.
3. **Define el rol estrecho y mueve el login a él,** primero en un entorno de staging que ejecute los mismos jobs, y luego en producción manteniendo a mano los privilegios antiguos durante un tiempo breve.
4. **Elimina los privilegios antiguos** y vigila las sentencias denegadas.

El costo es tiempo de calendario, no esfuerzo. Los equipos tienden a saltarse el segundo paso, y el resultado previsible es una ruta poco usada que se rompe meses después.

Drift

El **drift** es la distancia entre el acceso que declara el código y el acceso que tiene la base de datos. Aparece cuando un problema se arregla de noche con un `GRANT` escrito a mano: el grant funciona, el incidente termina y nadie lo anota. La comprobación compara lo declarado con lo real según un calendario, y debe fallar de forma ruidosa cuando no puede leer uno de los dos lados.

```

function checkRoleDrift():
    declared = loadDeclaredAccess()
    try:
        actual = readActualAccess()
        requests = loadAccessRequests()
        known = loadGatewayLogins()
    on ReadFailed:
        // an unreadable state is not a clean state
        notify("platform", "role drift check could not read a source")
        raise DriftCheckFailed

    for each extra in difference(actual.roleGrants,
declared.roleGrants):
        notify("platform", "grant not in code: " + extra)
    for each login in actual.loginsOwningObjects():
        notify("platform", "login owns objects: " + login)
    for each login in actual.membersOf("human_jit_write"):
        notify("platform", "login is a member of the ceiling: " +
login)
    for each login in actual.allLogins():
        if login is human and login not in known:
            notify("platform", "login unknown to the gateway: " +
login)
        // memberships must match what the login was issued for
        for each role in difference(actual.rolesOf(login),
expectedRoles(login)):
            notify("platform", "unexpected membership: " + role + "
for " + login)
        // a direct grant is legitimate only as part of a live request
        live = unexpiredTriples(requests, login)    // empty for
service logins
        for each g in difference(actual.directGrantsOf(login), live):
            notify("platform", "direct grant without a live request: "
+ g + " to " + login)
        for each login in actual.membersOf("human_breakglass"):
            if not hasOpenBreakGlass(requests, login):
                notify("platform", "break-glass member without a request:
" + login)

```

`expectedRoles(login)` es `human_readonly` para un login de sesión del gateway, además `human_breakglass` mientras haya una solicitud break-glass abierta, y el único rol `app_*` para el que el broker emitió el login. La comprobación detecta por tanto un `GRANT app_payments_service TO <session login>` o un `GRANT ledgerly_owner TO <login>` escritos a mano, además de un grant directo de privilegios.

Aquí, `roleGrants` cubre solo las membresías de rol a rol y los grants de privilegio a rol, que el código declara. Los logins no están en el código: van y vienen con las sesiones. Los logins humanos se comprueban contra

los registros del gateway. Los grants directos de cada login, incluidos los de servicio, se comparan con las tripletas de `access_request_privilege` que no han vencido. Un login de servicio no tiene solicitudes, así que cualquier grant directo sobre uno se señala; así se detecta un `GRANT` de un privilegio a un login de servicio escrito a mano de noche. Un grant que sobrevivió a su solicitud, o un miembro de break-glass sin una solicitud abierta, se señala incluso cuando el job de revocación informa de éxito. El acceso JIT funciona mediante grants directos y el vencimiento mediante un job de revocación; si el job se detiene, los grants permanecen, y esta comprobación lo nota con independencia del job (el Capítulo 8 cubre el emparejamiento). La comprobación lee solo grants y membresías. No ve lo que hace una sesión con ellos, que es de nuevo la parte Ejecución.

La comprobación no repara nada, porque revocar automáticamente grants desconocidos en producción es una nueva manera de romper producción de noche. Tampoco dice si el estado declarado es el correcto: un grant equivocado, correctamente declarado, pasa. Ese juicio corresponde a la revisión de accesos del Capítulo 14.

Trade-offs y costos

Cada control de este capítulo tiene un precio, y lo pagan personas concretas.

Los roles por workload cuestan mantenimiento. Cinco roles son cinco definiciones que mantener alineadas con cinco bases de código. Cuando un servicio necesita una tabla nueva, alguien actualiza el rol en el mismo cambio; si el grant llega después del código, el deploy falla. La mitigación es hacer que el grant forme parte de la migración. Con un servicio y dos ingenieros esta sobrecarga no es nada despreciable, y un esquema más pequeño, con un propietario, un rol de aplicación y un rol de solo lectura, es un primer paso razonable.

Los grants estrechos causarán sentencias denegadas. Algunas denegaciones son errores en los grants y otras son código que llegó más lejos de lo que debía. Quien está on-call paga ambas cuando una denegación aparece durante un incidente. Ejecutar staging con los mismos roles que producción reduce la sorpresa; ignorar el costo lleva a los equipos a ampliar un grant con prisa, y esa prisa es como empieza el drift.

Los grants a nivel de columna añaden carga de revisión. Cada lista de columnas debe mantenerse cuando cambia la tabla. Una columna sensible nueva queda expuesta por defecto bajo un grant a nivel de tabla, así que en las tablas sensibles concedo sobre columnas con nombre y acepto las líneas adicionales. Donde el motor carece de la función, la alternativa cuesta joins, tablas adicionales que migrar y una aprobación JIT más gruesa.

Las vistas y las reglas de fila añaden piezas móviles. Valen lo que pesan cuando lo importante es la forma o el aislamiento, y son un mal negocio cuando se añaden para sentirse cuidadoso. Una regla de fila cuyo contexto puede faltar es teatro de seguridad: añade fricción a cada consulta y falla en abierto en el caso para el que se escribió.

La disciplina de propiedad frena a algunas personas. Quienes están acostumbrados a crear una tabla desde una consola tienen que pasar ahora por una migración. La fricción es intencionada, y vale menos en una base de datos de desarrollo desechable que en producción. Aplica el modelo primero a producción, y relájalo en otros sitios solo cuando puedas decir a qué renuncias.

Cuándo no usar tanta estructura. Un prototipo con un solo desarrollador y sin datos de clientes no necesita el esquema de roles completo. La división entre propietario, aplicación y solo lectura basta hasta el segundo servicio o la primera columna sensible. Haz crecer el esquema cuando “¿quién ejecutó eso?” no pueda responderse desde el log, o cuando la credencial de un workload pueda hacer más daño del que justifica su trabajo.

Errores comunes

- **Tratar el rol de la aplicación como el propietario.** Es tentador porque así las migraciones se ejecutan al arrancar con un paso menos. El servicio pasa entonces a tener el poder de eliminar y volver a conceder. Usa un migrator aparte y un propietario sin login.
- **Conceder para igualar las consultas del año pasado.** Un rol copiado del login existente más amplio conserva todos los privilegios que alguna vez fueron cómodos. Deriva el rol de lo que el workload hace ahora, observado bajo su propio login.
- **Añadir un rol por persona.** Los roles son conjuntos de privilegios, y los logins son por persona. Un rol con el nombre de un individuo con-

vierte el esquema en una lista de personal. Mantén los roles centrados en los trabajos.

- **Usar un solo rol para varios servicios porque son “parecidos”.** Los servicios se van separando y el rol conserva la unión de lo que necesita cada uno. Un rol por servicio cuesta unas pocas líneas.
- **Dejar valores por defecto para las tablas futuras.** Un privilegio por defecto ahorra una línea en cada migración y expone la siguiente tabla sensible a un rol amplio. Mantén el valor por defecto vacío y concede de forma explícita.
- **Conceder el rol del techo a un login por una semana intensa.** El techo se convierte entonces en un grant permanente con un nombre amable. Mantén el techo sin asignar: solo lo lee la comprobación de solicitudes, y la prueba de drift señala cualquier membresía.
- **Ocultar una columna sensible tras una vista y dejar la tabla legible.** La vista no estrecha nada para un rol que puede llegar a la tabla base. Revoca sobre la tabla y prueba con un rol que no tenga ningún privilegio sobre la tabla.
- **Confiar en un esquema declarado que nadie compara.** Los roles en el control de versiones no detienen el `GRANT` de las 2 de la madrugada. Compara lo declarado con lo real según un calendario, y haz que la comparación falle de forma ruidosa cuando no pueda leer.
- **Usar el superusuario para el trabajo rutinario.** Funciona siempre, y por eso se extiende. Cuando una tarea rutinaria lo necesite, añade el privilegio que falta al rol correcto.

Ideas clave

- Un login compartido destruye a la vez la evidencia y el alcance: el log no puede decir quién actuó, y el login debe poder hacer lo que haga quien llame con el alcance más amplio.
- Mantén separados identidad, login y rol: un principal tiene una identidad, se conecta mediante un login, y el login es miembro de roles; nadie inicia sesión como un rol.
- `ledgerly_owner` es propietario de todo y no tiene login, `ledgerly_migrator` solo lo asume durante una migración, y cada workload recibe su propio rol `app_*` dimensionado a su trabajo.

- Empieza cada rol sin privilegios, usa grants a nivel de columna donde las columnas sensibles conviven con las ordinarias, mantén vacíos los privilegios por defecto y recurre a tablas de secrets aparte solo donde el motor no pueda conceder sobre columnas.
- `human_jit_write` es un techo que nunca se concede a un login: cada solicitud aprobada concede un subconjunto directamente al login de sesión, y ninguna solicitud puede incluir una columna Restricted.
- Los roles y los grants pertenecen al control de versiones y a pasos revisados del pipeline, y una comprobación de drift programada que falla de forma ruidosa cuando no puede leer la base de datos mantiene alineados el estado declarado y el real.
- Cada capa le cuesta algo a alguien: mantenimiento por los roles, sentencias denegadas por los grants estrechos, carga de revisión por las listas de columnas y una aprobación más gruesa donde el motor carece de grants a nivel de columna.

Los roles dicen qué puede hacer un principal una vez conectado, pero aún no dicen nada sobre cómo demuestra un servicio quién es. El Capítulo 4 da el siguiente paso: cómo llegan las credenciales a los workloads, por qué las credenciales dinámicas y de vida corta cambian la forma del problema y qué reemplaza a la contraseña en el archivo de configuración.

PARTE II

Acceso de las aplicaciones

Capítulo 4 — Credenciales y secrets

Imagina una tarde de viernes en Ledgerly. Alguien del equipo de backend ha pasado a otro grupo, y la persona de contacto de seguridad pregunta, con amabilidad, si se puede cambiar la contraseña de administrador compartida. Alex, del equipo de plataforma, está de acuerdo en que habría que hacerlo. Entonces Alex empieza a enumerar dónde vive la contraseña: la configuración de los cinco servicios, las variables del pipeline, el paso de export de `reporting-job`, un gestor de contraseñas del equipo y las conexiones guardadas en varios clientes de escritorio.

Cambiarla en la base de datos lleva un segundo. Cambiarla en todas partes, en el mismo momento, implica reiniciar todos los servicios de forma coordinada en horario laboral, y Robin, que está on-call este fin de semana, hace la pregunta razonable: ¿qué pasa con las facturas en curso? El cambio se aplaza a “una ventana tranquila” que nunca llega. La contraseña sobrevive a la persona, y luego a la siguiente.

Este capítulo responde a la pregunta que planteó aquel viernes: ¿cómo obtiene un servicio una credencial y cómo se cambia una sin provocar una caída? Añade el siguiente paso del estado objetivo del ejemplo que seguimos. Las credenciales de los servicios pasan a ser de corta duración o dinámicas, y llegan a los workloads por identidad en lugar de a través de un archivo compartido.

Las credenciales pertenecen a la parte Identidad del modelo de cinco partes, y son necesarias pero no son la respuesta. La tesis del libro es que “El acceso a bases de datos en producción no es un problema de distribución de credenciales; es un problema de autorización, ejecución y rendición de cuentas.” Manejar mejor las credenciales reduce quién puede obtener un login. No dice nada sobre lo que un login puede hacer una vez dentro, ni sobre lo que hizo, y de eso se ocupan los capítulos sobre grants, ejecución y registros.

El problema y por qué es difícil

Una **credencial** es lo que un principal presenta para autenticarse; un **secret** es cualquier valor que concede acceso si se revela. Una contraseña es ambas cosas. El Capítulo 3 dio a cada servicio su propio rol, de modo que un login dice lo que puede hacer. Este capítulo trata de la otra mitad: cómo demuestra el login quién es y cuánto tiempo sigue siendo válida esa prueba.

Una contraseña tiene un ciclo de vida aunque nadie lo diseñe. Se crea, se copia a cualquier lugar donde un servicio pueda leerla, se usa durante meses y al final se cambia o se olvida. Cada copia es un lugar donde una persona o un proceso puede leerla, y la pregunta “¿quién puede ver esta contraseña?” tiene tantas respuestas como copias. En el estado inicial la respuesta es “casi todo el equipo de backend, y cualquier cosa que pueda leer la configuración de un servicio”.

El remedio evidente es una contraseña mejor: más larga, aleatoria, guardada en un lugar más ordenado. Eso ayuda contra las adivinanzas, y adivinar nunca fue el riesgo principal. El riesgo es el número de lugares donde reside el valor, y el hecho de que ninguno puede vaciarse sin tocar los demás. Una contraseña fuerte en cinco archivos de configuración sigue siendo cinco copias.

La segunda dificultad es la **rotación**, que consiste en reemplazar una credencial por otra nueva de forma programada o por sospecha. La rotación es difícil porque una credencial es estado compartido entre dos sistemas: la base de datos conoce el valor, y también lo conoce cada consumidor. Si se cambia primero un lado, se rompe el otro. Un equipo que no puede cambiar una contraseña sin una caída dejará, con buen criterio, de cambiarla, y una política de rotación que nadie puede ejecutar añade una entrada al calendario y no elimina ninguna exposición.

Lo que las credenciales no pueden hacer

Toma un login del job de informes, válido durante una hora, filtrado en un canal de chat. Es una buena credencial según todas las medidas de este capítulo: es única, de corta duración y se obtiene por identidad. Durante esa hora todavía puede leer las tres columnas de `customers` que el rol tiene concedidas (`id`, `created_at` y `status`), en todas las filas, porque eso es

lo que `app_reporting_job` puede leer. Las duraciones cortas reducen la ventana; no reducen el rol. Tres cosas acotan una filtración, y ninguna es la credencial en sí: el **rol** que lleva (Capítulo 3), el **alcance de red** que decide desde dónde puede usarse (Capítulo 7) y el **log** que registra lo que hizo (Capítulo 14). Un login que solo puede presentarse desde la red del pooler, con un rol que no ve las columnas Restricted, es un incidente más pequeño de lo que cualquier política de contraseñas puede lograr.

Lo mismo vale para el mejor de los diseños que se describen a continuación. Un broker, un lease y la workload identity responden todos a quién puede obtener un login. No deciden lo que el login puede hacer, y no impiden que un login permitido ejecute una sentencia dañina.

Esto es una cuestión de ingeniería, no de política. Una regla que diga “rota cada 90 días” no dice nada sobre cómo se enteran los cinco servicios del nuevo valor en el mismo instante. La respuesta está en cómo se emiten, se entregan y se reemplazan las credenciales, que es un diseño que el equipo tiene que construir.

La vida de una credencial

Ayuda seguir una credencial desde su nacimiento hasta su retiro y preguntar, en cada parada, quién puede leerla y qué la revelaría. Los lugares habituales son estos.

Lugar	Quién puede leerla	Cómo suele filtrarse	Mejor lugar
Archivo de configuración en el repositorio	Todas las personas con acceso al repositorio, para siempre, a través del historial	Copiada a un fork, a un chat, a un ticket	Un gestor de secrets, referenciado por nombre
Variable de entorno	El proceso, sus hijos y cualquiera que pueda inspeccionar el proceso o sus volcados de fallo	Impresa por un endpoint de depuración o un informe de errores	Un archivo o memoria entregados al arrancar, con permisos de archivo estrechos
Imagen de contenedor o artefacto de build	Todas las personas que puedan descargar la imagen	Incrustada en el build, de modo que cada copia la lleva	Nada en la imagen; se obtiene al arrancar

Lugar	Quién puede leerla	Cómo suele filtrarse	Mejor lugar
Variable de CI/CD y log del job	Administradores del pipeline; a veces todos los jobs	Mostrada por un script, impresa por un comando que falla	Valor de corta duración obtenido por la propia identidad del job
Gestor de contraseñas del equipo	Todas las personas del vault compartido	Compartida por costumbre; sin registro de quién la leyó	Acceso individual con registro, para las pocas que quedan
Conexión guardada en un cliente de escritorio	Quien use ese laptop	Laptop perdido o con backup	Sin credencial guardada; credenciales de sesión desde un gateway (Capítulo 7)

Dos cosas destacan en la tabla. Primero, las filtraciones casi nunca son ingeniosas. Son copias hechas por comodidad: un valor pegado en un ticket para que un colega pueda reproducir un problema, una variable mostrada mientras se depuraba un deploy. Segundo, la columna “mejor lugar” es en su mayor parte la misma respuesta: el valor vive en un solo lugar gestionado, y los consumidores lo obtienen cuando lo necesitan.

Advertencia: Un secret que ha aparecido en una capa de imagen, en el historial de un repositorio o en el log de un job debe tratarse como revelado, aunque nadie lo haya leído. Borrar la línea no borra las copias. El remedio es reemplazar la credencial, y por eso un equipo necesita una vía de rotación que funcione antes del día en que haga falta.

Un secret tiene una propiedad más que conviene nombrar: cuánto tiempo sigue siendo válido. Supongamos que una contraseña es válida durante tres años: lleva tres años de exposición por cada copia que se haya hecho. Una copia antigua de una credencial que dura una hora no vale nada. Las duraciones son ilustrativas. Por eso la duración importa más que la fortaleza.

Gestores de secrets y cómo lee un servicio un secret

Un **gestor de secrets** es un servicio que almacena secrets, controla quién puede leer cada uno y registra cada lectura. HashiCorp Vault y los almacenes de secrets de los grandes proveedores de nube son ejemplos. El patrón importa más que el producto: un lugar guarda el valor, el acceso se decide por principal y cada lectura deja un registro.

Mover la contraseña a un gestor de secrets no reduce por sí solo el número de lectores. Cambia quiénes son y da nombre a cada lectura: una identidad de servicio concreta lee una ruta concreta, y el gestor de secrets dice cuándo. Es una ganancia real para la investigación y la revocación, y una ganancia modesta para la exposición.

El primer secret

Un servicio necesita algo con lo que demostrarse ante el gestor de secrets antes de poder leer nada. Si ese algo es otra contraseña almacenada, el problema se ha trasladado un nivel y no se ha resuelto nada. Esta es la pregunta del arranque, a veces llamada el problema del primer secret.

La respuesta es la **workload identity**: una identidad que la plataforma da a un workload en ejecución, como una instancia de servicio o un job, para que pueda autenticarse sin una contraseña almacenada. La plataforma que arranca el workload responde por él, y el gestor de secrets confía en esa afirmación. El servicio nunca tiene un valor de larga duración propio para este paso. Considero la workload identity lo que hace posible el resto del diseño, porque pone fin al regreso infinito.

Leer al arrancar y volver a leer

Un servicio tiene dos momentos razonables para leer su credencial de base de datos: una vez al arrancar, o cada vez que abre una conexión. Leer una sola vez es sencillo y funciona con credenciales estáticas que rara vez cambian. Falla cuando la credencial sí cambia, porque el proceso en ejecución conserva el valor antiguo hasta que se reinicia. Leer en el momento de la conexión, mediante un cliente que vuelve a preguntar al gestor de secrets tras un fallo de autenticación, soporta la rotación sin reinicio.

```

function connectWithCredential(service):
    attempts = 0
    while attempts < maxAttempts:
        try:
            slot = readSecret(path: "db/" + service + "/active")
            credential = readSecret(path: "db/" + service + "/" +
slot)
                return openConnection(host: "db-pooler.internal.example",
                    login: credential.login,
                    password: credential.password)
        on AuthenticationFailed:
            // the value may have rotated since the last read; read
again
            attempts = attempts + 1
        on SecretStoreUnavailable:
            // fail the start; never fall back to a value baked into
the image
            raise StartRefused
        raise StartRefused

```

El bloque muestra los dos fallos que importan. Un fallo de autenticación provoca una nueva lectura, de modo que una rotación no se convierte en una caída. Un gestor de secrets inalcanzable rechaza el arranque, porque recurrir a una copia de repuesto recrea el archivo que este diseño eliminó. Ese rechazo tiene un precio, y la sección Trade-offs y costos vuelve sobre él.

Nota: El propio gestor de secrets es una puerta. Quién puede leer la ruta `db/billing-api`, quién puede cambiarla y si las lecturas se registran son las mismas preguntas que para la base de datos. Dar a cada servicio acceso de lectura a cada ruta reconstruiría la contraseña compartida con un paso más.

Rotación sin caída

La cura para “cambia un lado primero y el otro se rompe” es no tener nunca un momento en que solo funcione un valor. Este es el **patrón de dos credenciales**, a menudo llamado rotación A/B. Cada servicio tiene dos logins, A y B, ambos miembros del mismo rol `app_*`. En cada momento uno está activo y el otro inactivo.

Para rotar, cambias el login inactivo, publicas el nuevo valor, indicas a los consumidores que cambien, esperas a que el login antiguo deje de usarse y entonces lo invalidas. En ningún momento un consumidor tiene un valor que la base de datos haya dejado de aceptar. En pseudocódigo, con los puntos de fallo marcados:

```
function rotateServiceCredential(service):
    active = readSecret(path: "db/" + service + "/active")    // "a" or
    "b"
    idle = otherSlot(active)
    if openSessions(loginFor(service, idle)) > 0:
        // an earlier rotation did not finish; changing this login
        cuts live sessions
        raise RotationIncomplete
    newPassword = generatePassword()
    // the idle login has no users, so changing it breaks nothing
    setLoginPassword(login: loginFor(service, idle), password:
newPassword)
    try:
        writeSecret(path: "db/" + service + "/" + idle,
                    value: { login: loginFor(service, idle), password:
newPassword })
        writeSecret(path: "db/" + service + "/active", value: idle)
    on SecretStoreUnavailable:
        // the active login is untouched; the idle one holds a value
nobody can read
        notify("platform", "rotation stopped before publish: " +
service)
        raise RotationAborted
        // consumers re-read on their next connection; wait for the old
login to drain
        waitUntil(openSessions(loginFor(service, active)) == 0, deadline:
drainDeadline)
        if openSessions(loginFor(service, active)) > 0:
            // the old value still works; the rotation is not finished
            notify("platform", "old login still in use after rotation: " +
service)
            raise RotationIncomplete
        setLoginPassword(login: loginFor(service, active), password:
generatePassword())
        auditLogAppend({ action: "rotate", subject: service, actor:
"rotation-job" })
```

Tres detalles sostienen el diseño. Primero, la base de datos y el gestor de secrets son sistemas separados sin transacción compartida, así que el orden de los pasos es lo que limita un fallo: el valor que está vigente nunca se cambia antes de que su reemplazo sea legible. Un fallo en cualquier paso deja funcionando la credencial anterior. Segundo, la comprobación de drenaje. La rotación solo termina cuando nada usa el login antiguo, y

un servicio que nunca vuelve a leer su secret se descubre en ese momento, no en una caída tres semanas después. Tercero, el último paso sobrescribe el login antiguo con un valor que nadie conserva, pero solo se ejecuta después del drenaje. Mientras alguna sesión siga teniendo el login antiguo, el valor antiguo funciona.

El costo es que ahora cada servicio tiene dos logins y un job que conoce ambos, y el consumidor debe volver a leer el secret tras un fallo. Un servicio que lee su credencial una sola vez al arrancar necesita un reinicio gradual para rotar, y ese reinicio es el punto en que una rotación A/B vuelve a convertirse en un evento de calendario. Rotar la contraseña de administrador compartida en el estado inicial es difícil por esta razón: no hay un segundo login, y ningún consumidor puede volver a leer nada.

Rotaría con una periodicidad que el equipo pueda sostener, no con la más corta que pueda escribir. Una sospecha de filtración es otra vía, descrita en “Cuando una credencial se filtra”.

Credenciales dinámicas y de corta duración

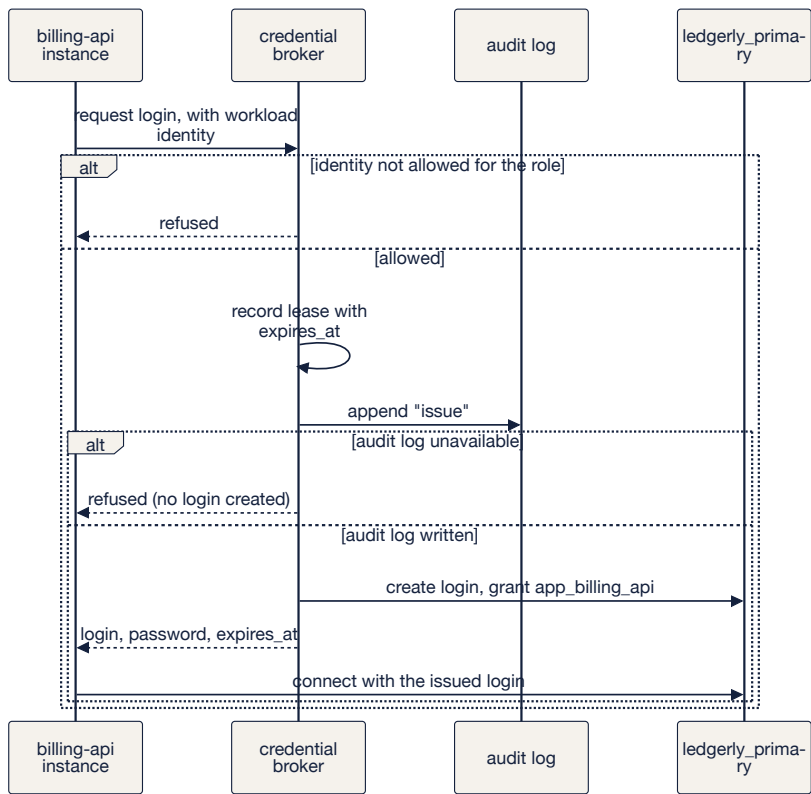
Una credencial estática rotada sigue teniendo una vida larga entre rotaciones. El siguiente paso es hacer que la duración sea tan corta que la rotación deje de ser un evento. Una **credencial de corta duración** vence por sí sola en un tiempo corto y definido. Una **credencial dinámica** es una credencial de corta duración generada bajo demanda para un principal, de modo que dos solicitudes nunca reciben el mismo valor. Todas las credenciales dinámicas son de corta duración; no toda credencial de corta duración es dinámica.

El mecanismo habitual es un **credential broker**: un servicio que, ante una solicitud, crea un login en la base de datos, lo hace miembro del rol `app_*` correcto y entrega la credencial del login al workload solicitante junto con un **lease**. Un lease es un registro que dice quién tiene la credencial, para qué rol y hasta cuándo. Cuando el lease vence, o alguien lo revoca, el broker elimina el login.

En Ledgerly el broker se conecta con su propio login, miembro de `ledgerly_access_admin`, el rol que crea y elimina logins y gestiona la pertenencia a roles. No posee tablas ni es el propietario, y nadie lo usa para trabajo con datos. Es un poder más estrecho que la propiedad, y aun

así es amplio: quien tiene el rol puede conceder cualquier cosa dentro de los roles que administra, incluidos los roles de aplicación y los humanos. El broker solo necesita una pequeña parte de ese poder. Darle menos supondría dividir el rol, lo cual tiene costos propios; aquí el broker usa el rol tal como está. El rol es uno solo, usado por tres principales, cada uno con su propio login: el paso del pipeline, este broker y el gateway. El Capítulo 3 explica qué acota ese poder. Para el broker, la protección es la única sentencia que se hace aquí: su credencial se guarda como la del propietario, solo su propia workload identity puede obtenerla, y cada acción que realiza va al registro de auditoría (Capítulo 14).

La secuencia siguiente muestra a una instancia de `billing-api` pidiendo un login. Fíjate en los dos puntos donde se rechaza la solicitud: antes de que exista nada, y cuando no se puede alcanzar el registro de auditoría.



En pseudocódigo, el orden de las operaciones es lo que mantiene correcta la secuencia. Nombres de función como `createLogin` son ilustrativos, como en el resto del libro.


```

function issueServiceLogin(request):
    identity = verifyWorkloadIdentity(request.proof)
    if request.role not in rolesAllowedFor(identity):
        raise IssueRefused
    password = generatePassword()
    login = identity.workload + "-" + randomSuffix()
    expiresAt = now() + ttlFor(request.role)
    // the lease comes first: the sweeper reads it, so no login can
    exist
    // without a record of when it must end
    leaseId = recordLease({ login: login, identity: identity.workload,
        instance: request.instance, role:
request.role, expires_at: expiresAt })
    try:
        auditLogAppend({ action: "issue", subject: login,
            actor: "credential-broker", lease: leaseId })
    on AuditLogUnavailable:
        markLeaseVoid(leaseId)
        raise IssueRefused
    // a failure here leaves a lease with no login; dropping a missing
    login is harmless
    createLogin(login, password: password, valid_until: expiresAt)
    grant(request.role, to: login)
    return { login: login, password: password, expires_at: expiresAt }

```

El lease se escribe antes de que exista el login, y la entrada de auditoría se escribe antes de crear el login. Si la base de datos falla después de registrar el lease, el sweeper encuentra un lease sin login y lo trata como ya revocado. Si el registro de auditoría está caído, no se emite ninguna credencial. Es una decisión deliberada, y significa que un registro de auditoría inalcanzable impide que arranquen instancias nuevas, un costo que el Capítulo 14 discute desde el otro lado.

Vencimiento y revocación

Un vencimiento solo impide que se autenticquen conexiones nuevas. Una sesión que ya está abierta sigue ejecutándose después de que la contraseña haya vencido, así que el vencimiento tiene dos partes: el login se invalida y se cierran las sesiones abiertas sobre él. Esa es también la diferencia entre rotación y **revocación**, que hace que una credencial o un grant dejen de funcionar ahora mismo. La rotación reemplaza; la revocación termina.

```

function revokeExpiredLeases():
  for each lease in expiredLeases(before: now()):
    try:
      closeSessions(login: lease.login)
      dropLogin(lease.login)
    on LoginNotFound:
      // the login is already gone, which is the goal
    on RevokeFailed:
      // an expired credential that still works is an incident
      notify("platform", "expired login still active: " +
lease.login)
      continue
    markLeaseRevoked(lease.id)
    try:
      auditLogAppend({ action: "revoke", actor: "lease-sweeper",
                        subject: lease.login, lease: lease.id })
    on AuditLogUnavailable:
      // the revoke is done; the log catches up from the retry
queue
      queueForRetry(lease.id)

function checkOverdueLeases():
  // scheduled separately, so one failing job cannot hide the other
  if count(expiredLeases(before: now() - graceMinutes)) > 0:
    notify("platform", "leases past expiry and not revoked")
  if now() - lastSuccessfulRun("revokeExpiredLeases") > maxRunAge:
    notify("platform", "lease sweeper has not completed recently")

```

La renovación sigue la misma disciplina. Una instancia en ejecución pide al broker que amplíe su lease antes de que termine, y el broker accede solo mientras el lease esté vigente y la duración total esté por debajo de un techo. Sin techo, la renovación convierte una credencial de corta duración en una de larga duración que casualmente se reemite. La duración en sí es una cuestión de criterio. No sé cuál es el número correcto para tu sistema, y depende de tres cosas: cuánto tiempo funciona una instancia entre reinicios, cuántas conexiones tiene cada instancia y cuánto tarda una renovación cuando el broker está bajo carga. A modo de ilustración, piensa en un lease de una hora renovado cada treinta minutos.

La ventaja frente a la rotación A/B es que una credencial filtrada vence por sí sola. El login nombra a la instancia, ya que se creó para esa instancia, de modo que el log de la base de datos responde a “¿quién ejecutó esa sentencia?” hasta el nivel del workload. Esto cumple lo que prometió el Capítulo 3: un login que corresponde a una única instancia de workload.

Workload identity y autenticación de base de datos basada en IAM

Hay una forma de eliminar por completo la contraseña almacenada. Algunas plataformas permiten que la base de datos confíe en el propio sistema de identidad de la plataforma: el workload pide a la plataforma un token de corta duración, y la base de datos acepta ese token en lugar de una contraseña. Los proveedores de nube ofrecen esto con nombres como autenticación de base de datos IAM, y algunas bases de datos pueden verificar tokens de un proveedor de identidad externo.

El atractivo es que no hay ningún secret que almacenar, obtener ni rotar. El token se emite para el workload cuando hace falta, es de corta duración (normalmente minutos) y está ligado a la identidad que la plataforma ya conoce. La revocación es tarea de la plataforma: quitar a la identidad el derecho a pedir tokens detiene los tokens nuevos, y los tokens ya emitidos pueden aceptarse hasta que venzan. El login en la base de datos sigue existiendo, sigue siendo miembro de su rol `app_*` y sigue apareciendo en el log.

Los costos son concretos:

- **Acoplamiento a la plataforma.** El diseño funciona donde la plataforma lo admite y en ningún otro sitio, así que un equipo con workloads en dos plataformas construye el patrón dos veces.
- **Correspondencia.** Alguien tiene que decir qué identidad de la plataforma corresponde a qué login y rol. Esa correspondencia es un grant, y merece la misma revisión que cualquier otro (Capítulo 14).
- **Conexiones abiertas.** En los motores habituales el token se comprueba cuando se abre una conexión. Como con cualquier vencimiento, una conexión abierta sobrevive a su token (véase “Vencimiento y revocación”), así que sigue haciendo falta un límite a la duración de la conexión (Capítulo 5).

Cuál de los tres enfoques usar depende menos de las modas que de lo que ofrezca la plataforma y de lo que el equipo pueda operar.

Enfoque	¿Secret almacenado?	Duración	Requiere	Costo principal
Credencial estática en un gestor de secrets, rotación A/B	Sí, un par por servicio	Hasta la siguiente rotación	Gestor de secrets, job de rotación	Los consumidores deben volver a leer; el job de rotación debe seguir funcionando
Credenciales dinámicas desde un broker	Solo el del propio broker	De minutos a horas, por instancia	Broker, <code>ledgerly_access_admin</code> , lease sweeper	El broker es un componente de gran valor y siempre disponible
Tokens de workload identity	Ninguno para la base de datos	Corta duración, normalmente minutos	Soporte de la plataforma y de la base de datos	Acoplamiento a la plataforma; los tokens solo se comprueban al conectar

Ninguno de estos enfoques elimina el problema de las credenciales; cada uno lo traslada. El login del propio broker es una credencial estática, guardada en el gestor de secrets, obtenida por la workload identity del broker y rotada A/B. El diseño con tokens depende de que el sistema de identidad de la plataforma sea correcto. Lo que cambia es cuántos secrets hay, cuánto tiempo es útil cada uno y quién puede leerlos.

TLS y verificación de certificados

Todo lo anterior trata de dónde se guarda una credencial y cuánto dura. Antes de llegar a la base de datos, cruza una red, y una conexión que no está cifrada lleva la contraseña, y los datos, en una forma que cualquier cosa en el camino puede leer. El cifrado en tránsito cierra eso, y es la base. Tiene que cumplirse en cada tramo. Una solicitud suele pasar por un pooler o un gateway antes de llegar a la base de datos, y cada uno de esos es una conexión aparte con su propio cifrado y su propia comprobación de certificado. Cifrar el tramo del cliente al pooler no dice nada sobre el tramo del pooler a la base de datos.

Hay una segunda tarea, más fácil de omitir: saber que el servidor es el correcto. Una conexión puede estar cifrada hacia un destino que no es la base de datos. La **verificación de certificados** consiste en que el cliente compruebe que el certificado del servidor lo emitió una autoridad en la que confía y que el certificado nombra al host al que quería llegar. Sin ella, el cifrado protege contra quien lee el tráfico y no hace nada contra quien responde a la conexión.

El ajuste que importa es el modo de verificación del cliente. La mayoría de los clientes de base de datos ofrecen un modo que cifra pero no verifica, y es popular porque funciona con cualquier certificado, incluidos los vencidos y los autofirmados. Considero ese modo la mitad del trabajo: oculta el tráfico a un observador pasivo y da una falsa sensación de que el servidor es el correcto. El remedio es distribuir la autoridad certificadora a cada servicio y verificar el nombre.

Eso trae sus propios costos. Los certificados vencen, y un certificado de servidor vencido es una caída con fecha exacta, así que el vencimiento necesita monitoreo y un procedimiento de renovación que se ensaye. Distribuir el certificado de la autoridad a cada consumidor es trabajo, y hay que actualizarlo cuando la autoridad cambia. Los certificados de cliente, donde el certificado es la credencial, solo funcionan con duraciones cortas si algo los emite automáticamente.

Cuando una credencial se filtra

Incluso con todos los controles anteriores, una credencial puede filtrarse, y el diseño debería decir de antemano qué ocurre a continuación. El primer paso es separar las dos preguntas que el pánico mezcla: ¿cómo evitamos que la credencial funcione y qué hizo mientras funcionó?

Para una credencial dinámica, la primera respuesta es una revocación: terminar el lease, cerrar las sesiones, y el login desaparece en el tiempo que tarde el broker en actuar. Para una credencial estática en una configuración A/B, una sospecha de filtración no es la rotación planificada. El job planificado espera a que el login antiguo se drene, y un valor filtrado sigue siendo válido mientras alguien mantenga una sesión con él. En una filtración, corta primero el login filtrado: quítale sus roles, cierra sus sesiones y cambia su contraseña sin esperar al drenaje, luego publica un valor nuevo

en el login inactivo y cambia el puntero activo. Las instancias que aún tienen el login antiguo pierden sus conexiones y fallan hasta que vuelven a leer el secret. Eso es un costo de disponibilidad, y lo pagaría: una credencial filtrada que sigue funcionando es el problema mayor. Un servicio que no puede volver a leer su secret necesita un reinicio, y el equipo debería saber cuáles son esos servicios antes del día en que importe. Para una contraseña de administrador compartida en el estado inicial, la primera respuesta es la caída con la que empezó este capítulo, que es el argumento más sólido para hacer el trabajo antes de un incidente.

La segunda pregunta la responde el log. Un login que corresponde a una única instancia de workload te permite ver las sentencias que ejecutó, las tablas que tocó y las horas en que estuvo activo. Eso reduce la revisión al radio de impacto de un rol en lugar de toda la base de datos. El procedimiento paso a paso, el orden en que actuar y cómo ensayarlo pertenecen al Capítulo 15, que trata del acceso durante los incidentes. Aquí lo importante es que un equipo que ha construido la revocación como un paso rutinario puede usarla en un mal día sin tener que inventarla.

Los regímenes de protección de datos y de tarjetas de pago suelen preguntar cómo se concede el acceso, cuánto dura y cómo se revoca. Los leases, los pasos de revocación y las entradas de auditoría descritos aquí son los registros que responden a esas preguntas.

El plan de migración de Ledgerly

El paso de la contraseña de administrador compartida a las credenciales de corta duración es una secuencia de pasos pequeños y reversibles. Los ordenaría según cuánto riesgo elimina cada paso y lo poco que puede romper.

1. **Haz un inventario de cada copia de la contraseña de administrador.** Servicios, variables del pipeline, el paso de export, el gestor de contraseñas, conexiones guardadas en herramientas cliente. La lista es el plan de trabajo. El inventario del Capítulo 1 es el punto de partida.
2. **Crea un login por servicio, en su rol `app_*`.** Usa los roles del Capítulo 3. En esta etapa las credenciales son estáticas, pero cada una es independiente, está guardada en el gestor de secrets y se lee por identidad.

3. **Saca los servicios del login de administrador de uno en uno.** Empezar por `reporting-job`, cuyo radio de impacto es el menor, luego `invoice-worker`, `admin-console` y `billing-api`. Dejar `payments-service` para el final, porque es el único servicio que llega a `payment_tokens` y `bank_accounts`, y un error ahí es lo que más cuesta. Cada traslado es un deploy que se puede revertir.
4. **Da a las migraciones y al paso de export sus propios logins.** El pipeline usa un login de corta duración en `ledgerly_migrator` (Capítulo 11); el rol en sí conserva privilegios permanentes de propietario, y solo la credencial es de corta duración. El paso de export usa `app_vendor_export` (Capítulo 13). Ninguno de los dos usa el login de administrador después de este paso.
5. **Pasa cada servicio a la rotación A/B.** Haz que cada servicio vuelva a leer su secret tras un fallo de autenticación, ejecuta la primera rotación en un período tranquilo y vigila la comprobación de drenaje. Un servicio que no se drena es un hallazgo, y el hallazgo es el objetivo de este paso.
6. **Introduce el broker para los servicios que se renuevan con facilidad.** Empieza con un servicio que se reinicia a menudo y mantiene pocas conexiones. Añade los demás a medida que el equipo aprenda cómo se comportan los leases. Mantén la A/B estática como fallback.
7. **Retira el login de administrador compartido.** Quítalo de cada configuración y del gestor de contraseñas. Una vez que la propiedad se haya trasladado a `ledgerly_owner` (Capítulo 3), retírale los privilegios y elimínalo, o desactívalo. Si el administrador resulta ser el superusuario del motor, como suele serlo el usuario maestro de un servicio gestionado, pasa a ser lo que el Capítulo 3 llama el instrumento sellado. Es una credencial que nadie usa en el trabajo diario. Se guarda donde el equipo de plataforma pueda recuperarla, y cada recuperación queda registrada. Cambia su credencial una sola vez, cuando todo el inventario esté despejado.

Un equipo podría detenerse tras el paso cinco y estar mucho mejor: cada servicio tiene su propia credencial, la lee a través de una identidad y la rota sin una caída. El broker es un añadido para equipos que pueden operarlo.

Trade-offs y costos

Cada movimiento de este capítulo traslada el costo de un incidente a una rutina. Aquí es donde la rutina se encarece.

Dependencia del gestor de secrets. Una vez que las credenciales provienen de un gestor de secrets, un gestor de secrets inalcanzable impide que arranquen instancias nuevas y, con leases cortos, acaba impidiendo que las instancias en ejecución se renueven. Eso lo convierte en parte de la disponibilidad de la ruta a la base de datos. Opéralo con el cuidado que das a la base de datos: más de una instancia, una recuperación probada y una rotación de on-call que sepa que existe. Guardar en cache una credencial durante la duración de un lease suaviza una caída breve, y alarga el tiempo durante el que una credencial revocada aún puede usarse.

Tormentas de conexiones. Si todas las instancias de un servicio se renuevan en el mismo minuto, o se reinician juntas tras un deploy, la base de datos ve una ráfaga de conexiones y autenticaciones nuevas. La autenticación no es gratuita, y una ráfaga provocada por el propio sistema puede ralentizar la base de datos para todos los demás roles, una tormenta de conexiones sin ninguna causa externa. Escalona las renovaciones con jitter, mantén estable el pool de conexiones entre renovaciones y deja que el Capítulo 5 determine cuántas conexiones puede mantener un rol.

Depuración más difícil. Un login que existe durante una hora ya no está cuando alguien lee el log a la mañana siguiente, y un nombre como `billing-api-4f2a` no significa nada sin el registro del lease. Conserva los leases y su identidad durante todo el tiempo que se conserve el registro de auditoría, y da a quien esté de on-call una búsqueda de login a instancia. Si falta esa búsqueda, la credencial de corta duración hace la investigación más lenta, no más rápida.

Un broker de gran valor. El login del broker puede crear logins en cada rol de aplicación, así que merece el esfuerzo de un atacante. Mantén su código pequeño y revisado, y dale la atención que das a lo que protege; un broker que recibe menos es un riesgo en sí mismo.

Cuándo no usar esto. Un equipo con un solo servicio, un puñado de ingenieros y un almacén de secrets gestionado puede estar bien servido con una credencial estática independiente, guardada en el gestor de secrets y rotada A/B a mano dos veces al año. Un broker es un servicio más que

operar y un sistema más que el equipo debe aprender; por debajo de cierto tamaño, las piezas móviles añadidas causarán más caídas de las que eviten. Si no puedes nombrar a quién se enviaría un page cuando el broker esté caído, esperaría. Los tokens de workload identity son el paso más barato cuando la plataforma los ofrece, y aun así requieren que se revise la correspondencia.

Errores comunes

- **Tratar el gestor de secrets como la respuesta completa.** Es tentador porque el cambio es visible y rápido. La contraseña compartida sigue siendo compartida si cada servicio lee la misma ruta. Da a cada servicio su propia ruta y su propio login, y revisa quién puede leer cada una.
- **Una rotación que nadie consume.** Un job rota el valor, pero los servicios lo leen al arrancar y nunca más, de modo que el valor nuevo queda sin leer hasta que se retira el antiguo y algo falla. Prueba que los consumidores recogen una rotación, y haz que la comprobación de drenaje forme parte del job.
- **Recurrir a un valor incrustado.** Una copia de “última configuración conocida buena” en la imagen hace confiable el arranque y recupera el archivo que el diseño eliminó. Rechaza el arranque, genera una alerta y repara la dependencia.
- **Cifrar sin verificar.** Activar TLS satisface una lista de verificación y el modo que solo cifra hace que la conexión funcione. Verifica el certificado y el nombre del host, y monitorea la fecha de vencimiento.
- **Duraciones sin techo.** Una renovación sin límite reconstruye una credencial de larga duración. Limita la duración total de un lease y haz que la instancia se vuelva a autenticar con su identidad.
- **Tratar el rol del broker como pequeño.** `ledgerly_access_admin` es más amplio que la tarea del broker, así que un broker que sea tomado puede llegar tan lejos como los roles que administra. Restringe qué principales pueden llamarlo y qué roles `app_*` puede pedir cada uno, y revisa sus entradas de auditoría. Trata el rol amplio como un costo que has aceptado, no como un límite.

Ideas clave

- Cada copia de una credencial es un lector, así que el cambio más útil es reducir el número de copias y dar nombre a cada lectura que queda.
- La rotación es difícil porque una credencial es estado compartido; dos logins por servicio permiten cambiar uno mientras el otro sigue funcionando.
- Una rotación que los consumidores no recogen, o que nadie ejecuta, es una entrada de calendario y no un control, así que la comprobación de drenaje y una alerta sobre un job de rotación que ha dejado de ejecutarse forman parte del diseño.
- Las credenciales dinámicas dan a cada instancia de workload su propio login con un final incorporado, al precio de un broker, un registro de leases y una dependencia del gestor de secrets.
- La workload identity elimina la contraseña almacenada donde la plataforma lo admite, y traslada la confianza a la plataforma y a la correspondencia entre identidad y login.
- Verificar el certificado y el nombre del servidor es lo que hace que TLS proteja la credencial; el cifrado sin verificación oculta el tráfico y deja sin comprobar la identidad del servidor.
- Cuando una credencial se filtra, la rapidez de la revocación y la claridad del log deciden cuán contenido queda el evento, y eso depende del trabajo hecho antes de la filtración.

Las credenciales deciden quién entra; no dicen nada sobre cuántas conexiones abre ni cuánto tiempo puede una sentencia retener el primary. El Capítulo 5 convierte la conexión misma en una política: límites por rol, pooling, timeouts y las etiquetas que facilitan encontrar una consulta descontrolada.

Capítulo 5 — Las conexiones como política

Piensa en Ledgerly una noche tranquila de martes. Alguien del equipo aumentó el paralelismo de la agregación nocturna de `reporting-job` para que terminara antes de la mañana, un cambio que en la revisión parecía inofensivo. A la 01:40 el job arranca, abre una conexión por cada uno de sus hilos de trabajo y empieza a leer `payments` e `invoices` del `primary`, con el login compartido `admin`. En pocos minutos `billing-api` empieza a devolver errores a clientes de una zona horaria en la que ya es de mañana. A Robin, que está on-call, le llega un page por una caída que afecta a clientes.

Robin abre la base de datos y descubre que nada está roto. El `primary` está activo, la actividad de disco es normal y ninguna sentencia ha hecho nada incorrecto. El servidor tiene un número fijo de slots de conexión, el informe se ha llevado la mayoría y la API espera en la puerta. Como todos los que llaman usan el mismo login, la base de datos no puede decir que el informe es la causa. Solo puede decir que los slots están llenos.

Este capítulo responde a una pregunta que se sitúa entre la identidad y los datos: ¿cómo hacemos que la propia conexión sea un lugar donde se aplique la política? En el modelo de cinco partes del Capítulo 1 actúa sobre dos de ellas. La conexión es la Ruta, porque el pooler decide cómo llegan los servicios y cuántos pueden hacerlo. También es la primera cota de la Ejecución, porque los límites y los timeouts dicen cuánto puede tomar de todos los demás un solo cliente, y una sola sentencia. El paso del ejemplo que seguimos es que las conexiones quedan moldeadas por un pooler y por límites y timeouts por rol.

El problema y por qué es difícil

Una conexión es un recurso antes que cualquier otra cosa. En la mayoría de los motores cada una cuesta al servidor memoria y una parte de su planificación, y el servidor permite solo un número fijo de ellas. El límite es una propiedad del servidor, no de quien llama, así que el primero que pide muchas consigue muchas. Nada en la configuración por defecto dice que el informe nocturno deba tener menos slots que la API de cara al cliente.

Una contraseña no toca nada de esto. Decide quién entra; no dice nada sobre cuántos slots ocupa ni durante cuánto tiempo. Por eso un libro que trata de algo más que de distribuir credenciales necesita un capítulo sobre conexiones.

Tres hábitos empeoran el problema. Primero, el número de conexiones es un producto, no una suma: las instancias de un servicio multiplicadas por el tamaño del pool de cada instancia, multiplicado de nuevo cada vez que un autoscaler añade una instancia. Supongamos que un equipo fija un pool de veinte en un servicio que escala hasta diez instancias. Ha pedido doscientas conexiones, y probablemente no lo decidió a propósito. Segundo, una conexión se mantiene mientras dura su sentencia, así que una consulta lenta ocupa un slot todo el tiempo, y los slots son lo que necesita el siguiente cliente. Tercero, quien falla tiende a insistir con más fuerza. Un cliente que agota el timeout y reintenta de inmediato convierte una base de datos lenta en una multitud en la puerta.

La respuesta evidente es subir el límite del servidor. Ayuda durante una semana y luego hace mayor el siguiente incidente. Más conexiones significan más trabajo concurrente sobre el mismo primary, y la base de datos no se vuelve más rápida ejecutando cien sentencias a la vez porque se permiten cien.

Los controles viven en varios lugares que nadie gestiona en conjunto: la definición del rol, el pool de la aplicación, un posible pooler y los ajustes que deciden cuánto puede ejecutarse una sentencia. La disponibilidad es una de las propiedades en juego en el modelo de amenazas del Capítulo 2, y los accidentes la dominan. Un login compartido convierte cada accidente en el accidente de todos.

En el estado inicial, entonces, los cinco servicios y cada ingeniero conectado por VPN toman de un único conjunto de slots del servidor bajo un mismo nombre. No hay timeouts más allá de lo que cada biblioteca cliente tenga por defecto, y no hay registro de qué cliente tiene qué conexión. Las secciones siguientes lo sustituyen por un presupuesto por rol, un pooler que lo aplica, timeouts que fijan un límite superior al tiempo que cualquiera puede retener un slot y una etiqueta en cada conexión que dice quién es su responsable.

Un presupuesto de conexiones por rol

El primer paso es el más barato. El Capítulo 3 dio a cada servicio su propio rol de aplicación, lo que significa que por fin se puede asociar un límite a un nombre. Un límite por rol dice: este principal puede mantener como máximo este número de conexiones, sin importar cuántas instancias ejecute ni con cuánta insistencia pida. Cuando alcanza el límite, se rechazan las conexiones nuevas de ese rol, y todos los demás roles siguen funcionando.

Ese rechazo es la política. Traslada el fallo de “la base de datos está llena para todos” a “este cliente se ha quedado sin slots”, y lo segundo es un page mucho mejor para que lo reciba Robin. El radio de impacto de un cliente voraz pasa a ser el propio cliente, y el rechazo nombra al cliente, que es la mitad de rendición de cuentas del mismo paso.

Los números necesitan razonamiento, no una fórmula. Trabajaría desde el límite del servidor hacia abajo. Primero se resta una reserva que nada más puede usar, porque las personas y herramientas que reparan el sistema también necesitan conectarse. La reserva cubre las sesiones de la plataforma y los logins de los tres principales que usan `ledgerly_access_admin` (el paso del pipeline que aplica las definiciones de roles, el credential broker y el access gateway). Después se divide el resto entre los roles de aplicación según lo que necesita cada uno en su punto de máxima actividad, más algo de margen. La tabla siguiente es una ilustración para el primary. Los números son hipotéticos, no una recomendación.

Principal	Rol	Límite de conexiones (ilustrativo)	Por qué este tamaño
billing-api	app_billing_api	80	El servicio más ocupado; atiende la aplicación web y la API pública
invoice-worker	app_invoice_worker	30	Dirigido por cola; la concurrencia la fija el número de workers
payments-service	app_payments_service	30	Estable; habla con proveedores externos entre sentencias
admin-console	app_admin_console	15	Acciones estrechas y auditadas de unas pocas personas
Migraciones	ledgerly_migrator	3	Una migración a la vez, desde el pipeline
Sesiones humanas en el primary	Logins de sesión de tipo JIT y break-glass, creados por el gateway	8	Solo se ejecutan aquí las sesiones con una solicitud aprobada (Capítulo 6); las lecturas rutinarias van a ledgerly_replica_2. Los roles human_jit_write y human_breakglass no llevan límite, porque ningún login es miembro del primero y el segundo se concede por sesión
reporting-job	app_reporting_job	10	Solo en la transición: lee el primary en el estado inicial y durante el traslado, y la fila pasa a cero en el estado objetivo
Reserva	administración de accesos y plataforma	10	Solo para reparación y para emitir logins
Sin asignar	ninguno	14	Margen, para que un error de redondeo no sea una caída

Supongamos que el servidor permite 200 conexiones. Las filas asignadas suman 186, y las 14 restantes se dejan sin asignar deliberadamente. Las sesiones humanas rutinarias no están en esta tabla porque no se ejecutan en el primary. Su presupuesto se fija en ledgerly_replica_2, adonde van las lecturas rutinarias; digamos, 20 slots allí. La fila de reporting-job existe solo mientras el informe siga leyendo el primary. En el estado objetivo lee de ledgerly_replica_2, con su propio presupuesto en ese host, fi-

jado en la capa de conexión del pooler (el Capítulo 6 cubre el traslado). Las definiciones de los roles podrían llevar los límites en una notación como esta, que es ilustrativa y no pertenece a ningún motor:

```
// illustrative notation: the limit belongs to the role, not to a
person
ROLE LIMIT app_billing_api      CONNECTIONS 80;
ROLE LIMIT app_invoice_worker  CONNECTIONS 30;
ROLE LIMIT app_payments_service CONNECTIONS 30;
ROLE LIMIT app_admin_console   CONNECTIONS 15;
ROLE LIMIT ledgerly_migrator   CONNECTIONS 3;
ROLE LIMIT app_reporting_job    CONNECTIONS 10;
```

Nota: Los motores difieren en aquello a lo que se asocia un límite o un ajuste. Algunos cuentan las conexiones por rol, otros por login y otros solo por base de datos o por servidor. En algunos motores, un límite o un valor por defecto escrito en un rol se aplica solo a las conexiones que inician sesión como el propio rol, y no lo heredan los logins que son miembros de él. El Capítulo 4 da a cada instancia de servicio su propio login dinámico, miembro del rol `app_*`, así que en esos motores el límite a nivel de rol y los timeouts a nivel de rol que siguen no se aplicarían, sin avisar.

Hay dos lugares que pueden llevarlos en su lugar: el broker, que puede fijar un límite por login y los mismos ajustes al crear cada login, y el pooler, que puede llevar los ajustes pero quizá no pueda sumar los logins de un rol en un total. La sección del pooler que sigue expone tres formas de obtener un total por rol cuando no puede.

Dos propiedades hacen útil la tabla. Los límites se comprueban donde empieza la conexión, así que quien llama se entera del rechazo de inmediato y no tras la primera sentencia lenta. Y la reserva queda fuera de todos los roles de aplicación, de modo que en la peor noche el equipo de plataforma todavía puede entrar a arreglar las cosas. Un presupuesto que llena todo el servidor tiene el mismo defecto que una contraseña de administrador compartida: funciona hasta el momento en que alguien lo necesita más.

Las sumas no tienen que ser conservadoras. Si `billing-api` y `payments-service` rara vez coinciden en su pico, puedes dejar que los límites de las aplicaciones sumen más de lo que el servidor permite, y aceptar que dos

picos simultáneos podrían chocar. Solo lo haría con la reserva apartada, y solo después de que el equipo haya observado el uso real durante un tiempo. Lo normal es un presupuesto que sume, porque un presupuesto que no puede violarse es más fácil de razonar a las tres de la madrugada.

Pooling: dónde se comparten las conexiones

Un pool mantiene un conjunto de conexiones abiertas y las presta a quienes llaman, de modo que cada solicitud no paga por una conexión nueva y la base de datos ve un número acotado de ellas. Hay dos lugares donde ponerlo.

Casi todos los servicios ya tienen un pool en su biblioteca de base de datos, y debería dimensionarse según el trabajo que una instancia puede hacer a la vez, no según el tráfico que espera atender. Un pool pequeño que hace esperar brevemente a quien llama es más amable con la base de datos que uno grande, y debería rendirse tras una espera corta con un error que el servicio pueda convertir en una respuesta clara. Empieza pequeño, observa cuánto esperan los clientes y sube el tamaño con evidencia.

El pooler delante de la base de datos

Un **pooler** es un proxy de base de datos cuya tarea principal es aceptar muchas conexiones de clientes y multiplexarlas sobre un número menor de conexiones de servidor. PgBouncer y ProxySQL son ejemplos habituales. El servicio se conecta al pooler como si fuera la base de datos, y el pooler decide cuándo se presta una conexión real.

La pregunta que el libro no deja de hacer se aplica aquí también: ¿qué nombre ve la base de datos? El Capítulo 4 dio a cada instancia de servicio su propio login dinámico, creado por el credential broker y miembro del rol `app_*` del servicio. El pooler tiene que encajar en ese modelo, y yo lo construiría con esta forma. La instancia presenta su propio login y su credencial al pooler. El pooler los verifica contra la base de datos y mantiene las conexiones de servidor bajo ese mismo login. No tiene un almacén de contraseñas propio, ni un login propio que compartan todos los clientes.

Una credencial rotada o vencida llega al pooler a través de la base de datos, no a través del pooler. Cuando un lease termina o se revoca, el broker elimina el login (Capítulo 4), de modo que el siguiente intento del pooler

de autenticarse o de abrir una conexión de servidor bajo ese login es rechazado, y cierra las conexiones de servidor inactivas que mantiene para él.

Cómo se sigue cumpliendo un límite por rol no tiene una respuesta universal, y no supondría que tu pooler pueda darla. Muchos poolers mantienen sus límites por par de login y base de datos, y algunos añaden un límite por login o por base de datos. No conozco ninguna capacidad general que sume las conexiones de muchos logins distintos en un tope a nivel de rol, así que lee la documentación de tu pooler en lugar de fiarte de este capítulo. Si el tuyo no puede hacerlo, el total por rol tiene que salir de otra parte. Hay tres lugares, y cada uno tiene su precio.

- **El pooler se conecta aguas arriba con un login estable por rol.** La credencial dinámica de la instancia solo autentica al cliente ante el pooler, y el límite por login del pooler es exactamente el límite del rol. El precio es el modelo de identidad del Capítulo 3:
 - El log de la base de datos muestra un nombre por rol, así que la atribución a una instancia pasa al log del pooler y a la etiqueta de aplicación descrita más abajo.
 - El pooler debe verificar por sí mismo las credenciales dinámicas, lo que lo convierte en consumidor del estado de leases del broker.
 - También guarda una credencial de larga duración por cada rol, que hay que proteger y rotar A/B como describe el Capítulo 4.
 - Revocar el login de una instancia en el broker ya no cierra sus conexiones de servidor; el pooler tiene que terminar las sesiones de ese cliente.
- **La base de datos aplica un límite por rol, donde el motor lo admite para logins miembros.** Es exacto, está donde nada puede eludirlo y los logins de instancia y su atribución quedan intactos. El precio es que el soporte varía, como dice la nota anterior. Donde funciona, el rechazo viene de la base de datos después de que la conexión ya ha costado una autenticación, y nada se multiplexa.
- **El broker limita cuántos logins emite por rol.** Con un límite por login fijado al crear cada login, el total del rol no puede superar el número de logins por ese límite. El precio es que esto es una cota superior y no un conteo exacto, y que el tope tiene que coincidir con el te-

cho de instancias del autoscaler. Un escalado por encima del tope se convierte en un deploy que no puede obtener credenciales, lo cual es un problema de disponibilidad, y las renovaciones que solapan brevemente un login antiguo y uno nuevo necesitan espacio bajo el tope.

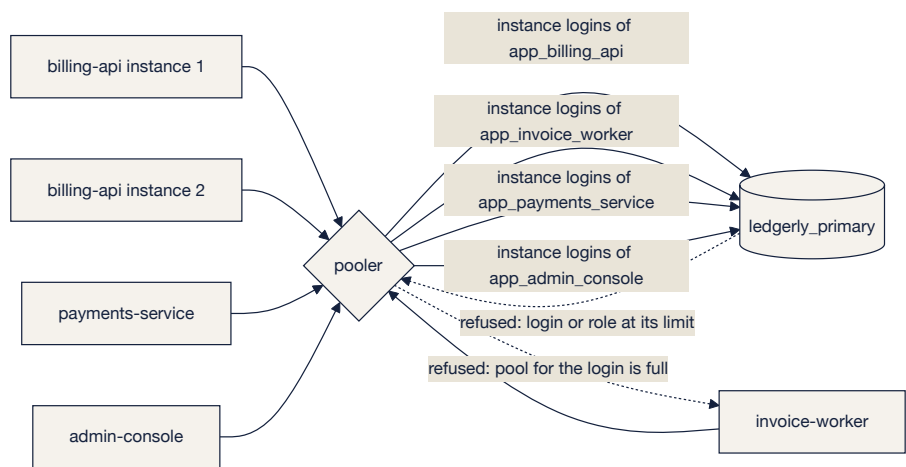
Probaría primero el límite del lado de la base de datos, donde el motor lo respeta para logins miembros. Donde no lo haga, el tope del broker conserva el modelo de identidad y cuesta un techo que mantener. Elegiría el único login aguas arriba por rol solo cuando el número de conexiones obligue a multiplexar entre instancias, y dejaría por escrito lo que el log ya no dice. Las tres son mejores que el diseño que asigna a cada cliente de cada rol un único login compartido. Eso recrea la cuenta de administrador compartida una capa más abajo: el log vuelve a mostrar un solo nombre, y un límite se aplica a todos a la vez.

En la forma pass-through recomendada el pooler logra poca multiplexación. En modo transaction los clientes de una instancia comparten las conexiones de servidor de esa instancia, pero dos instancias no comparten entre sí, así que el total de un rol es al menos el número de instancias vivas por un pequeño mínimo. Lo que el pooler aporta son otras tres cosas.

- **Rechazo central.** Un solo lugar dice que no, por login y por pool, de la misma manera para cada biblioteca cliente.
- **Los mismos límites para cada servicio.** Los tamaños de pool y las esperas de cada servicio dejan de ser la única línea de defensa.
- **Un punto de estrangulamiento de red.** Los servicios entran a la base de datos solo a través del pooler. La política de red, que dice qué hosts pueden abrir una conexión TCP y que el Capítulo 6 define y el Capítulo 7 escribe, admite los hosts del pooler y a nadie más, de modo que una credencial filtrada de un servicio es inútil desde cualquier lugar al que el pooler no sea alcanzable. El alcance acota una filtración, como dice el Capítulo 4, y el pooler es donde esa cota es fácil de enunciar.

El costo es un sistema nuevo en la ruta, tratado en los trade-offs más abajo. Para un servicio pequeño con una sola instancia bastan el pool de la aplicación y un límite de rol, y no vale la pena añadir el pooler hasta que el número de instancias o la necesidad de un único punto de estrangulamiento lo indiquen.

El diagrama siguiente muestra la forma pass-through, que deja los logins de instancia visibles para la base de datos. Fíjate en los dos puntos donde se rechaza a quien llama: en el pooler, cuando el pool de un login está lleno, y en la base de datos, cuando un login o, donde el motor lo admite, el rol está en su límite.



Modo session y modo transaction

Un pooler presta una conexión de servidor durante un período, y la duración de ese período es su **modo**. En modo session el cliente conserva la misma conexión de servidor desde que se conecta hasta que se desconecta; el pooler no ahorra nada salvo el costo de conectar. En modo transaction el cliente retiene una conexión de servidor solo durante una transacción, y la siguiente transacción puede ejecutarse en otra distinta. El modo transaction multiplexa muchos más clientes sobre pocas conexiones, y por eso resulta atractivo.

Viene con un costo fácil de pasar por alto, porque afecta a la autorización y no al rendimiento. Todo lo que un cliente establece en su sesión vive en una conexión de servidor, y en modo transaction el cliente no es propietario de ninguna. Los ajustes hechos fuera de una transacción pueden desaparecer en la sentencia siguiente o, peor, pueden quedarse en una conexión que el pooler presta a otra persona. Entre las cosas que dependen del estado de sesión están las tablas temporales, las prepared statements en muchas configuraciones, los advisory locks, los timeouts a nivel de sesión y cualquier valor de contexto que tu propio diseño almacene en la sesión.

Este último punto toca directamente al libro. En el Capítulo 3, una regla de fila puede depender de un valor como `current_case_customer()`, y `app_admin_console` lleva la identidad de un principal humano al registro de auditoría y a `audit_events`. Si cualquiera de los dos valores se fija una vez por sesión y se lee después, entonces en modo transaction puede faltar o, en el peor caso, pertenecer al cliente anterior.

Advertencia: En modo transaction, un valor de contexto fijado a nivel de sesión puede quedarse en una conexión de servidor y ser leído por la transacción del siguiente cliente. Una regla de fila o un registro de auditoría que lo lea puede entonces actuar por el cliente equivocado o la persona equivocada. Fija esos valores dentro de la transacción, acotados a ella, y haz que el código que los lee falle cuando el valor esté ausente.

El patrón que se sostiene es fijar el contexto dentro de la transacción y dejar que termine con la transacción. El bloque siguiente muestra la forma, y está escrito para seguir siendo correcto cuando el trabajo falla a medias.

```
function runAsPrincipal(principal, customerId, work):
  transaction:
    // transaction-scoped: it disappears at commit or rollback, so
    // the next client cannot inherit it from this connection
    setLocalContext("acting_principal", principal.id)
    setLocalContext("case_customer", customerId)
    try:
      work()
    on ContextMissing:
      // a row rule or audit write found no value: stop, do not
      guess
      raise ActionRefused
    // the commit happens here, once, after all of the work
  succeeded
  // on any error the transaction rolls back, and the context goes
  with it
```

El log de la base de datos sigue mostrando el login de la instancia, pero en modo transaction los clientes de una instancia comparten sus conexiones de servidor, así que el log no puede decir cuál de los hilos o solicitudes de la instancia tenía un slot. Para averiguarlo necesitas la vista propia que el pooler tiene de sus clientes, y hay que planificarla antes del primer incidente. Por eso importan las secciones siguientes sobre timeouts y nombres.

Las personas no usan este pooler. Llegan a la base de datos a través del access gateway, que crea un login para cada sesión (Capítulo 7). Sus límites son el tema de la última parte de la sección siguiente, porque una persona puede hacer algo que un servicio no puede: levantar el límite.

Los timeouts como política

Un slot se presta mientras dura una sentencia o una transacción, y en la mayoría de los motores nada termina ninguna de las dos por defecto. Los timeouts son la forma en que el equipo de plataforma dice cuánto tiempo puede retener un recurso un rol, y son la parte de este capítulo que acota la Ejecución: una sentencia que se ejecuta demasiado se cancela donde se ejecuta, y el slot vuelve. Pertenecen a la definición del rol y a la configuración del pool, y no a la buena voluntad de cada biblioteca cliente, por lo que este capítulo los trata como política.

Tres timeouts pertenecen a la base de datos y dos al pool.

Timeout	Perte- nece a	Qué detiene	Quién lo nota
Sentencia	Base de datos	Una consulta que se ejecuta mucho más de lo que el rol debería necesitar	Quien llamó con esa sentencia
Espera de bloqueo	Base de datos	Una sentencia en cola tras un bloqueo, y todos los que esperan detrás de ella	Quien llama; también cualquiera que espere sobre la misma fila
Inactividad en transacción	Base de datos	Una transacción abierta y luego abandonada, que retiene bloqueos y slots	La aplicación que la abrió
Duración de la conexión	Pool	Una conexión que permanece abierta durante días y conserva su credencial o estado antiguos	El pool, que debe reconectar
Espera de adquisición del pool	Pool	Quien llama esperando indefinidamente una conexión libre	El servicio que atiende la solicitud

Los distintos roles necesitan valores distintos, porque hacen trabajos distintos. Una solicitud en `billing-api` es una persona esperando una página; una sentencia que necesita treinta segundos ahí es un bug. `invoice-worker` puede ejecutar legítimamente una sentencia más larga mientras

construye un lote. `reporting-job` ejecuta lecturas largas por diseño, en una réplica donde una lectura larga no bloquea a los que escriben. Un valor por defecto compartido sería incorrecto para todos ellos, así que los valores se asocian al rol. El bloque siguiente usa una notación ilustrativa, y los valores son hipotéticos.

```
// illustrative notation: defaults for the role; see the note above on
// members, because the broker or the pooler may have to apply them
per login
ROLE SETTING app_billing_api          statement_timeout 5s, lock_wait 2s,
idle_in_transaction 10s;
ROLE SETTING app_invoice_worker       statement_timeout 30s, lock_wait 5s,
idle_in_transaction 30s;
ROLE SETTING app_reporting_job        statement_timeout 15m, lock_wait 5s,
idle_in_transaction 60s;
ROLE SETTING human_readonly           statement_timeout 60s, lock_wait 5s,
idle_in_transaction 5m;
ROLE SETTING ledgerly_migrator        statement_timeout none, lock_wait
3s, idle_in_transaction 60s;
```

La última línea refleja una lección de migraciones que cubre el Capítulo 11: un cambio de esquema puede ejecutarse durante mucho tiempo, pero debería esperar solo brevemente un bloqueo.

Dos reglas de orden importan más que los valores individuales. El timeout de sentencia de la base de datos debería ser más corto que el deadline de solicitud de la aplicación, porque si la aplicación se rinde primero abandona una sentencia que sigue ejecutándose y el slot permanece ocupado; y la espera de adquisición del pool debería ser la más corta de todas, para que quienes llaman se enteren pronto de la congestión. Cuando un deadline salta primero, la conexión aún puede llevar una sentencia en ejecución, así que el servicio la cierra y no se la presta al siguiente cliente.

La transacción inactiva es la sorprendente. Un servicio que abre una transacción, llama a un proveedor de pagos externo y espera con la transacción aún abierta deja que la lentitud del proveedor retenga bloques y un slot en la base de datos. La solución a largo plazo es mantener las llamadas externas fuera de las transacciones, lo cual es un cambio de código; el timeout mantiene acotado el costo mientras ese cambio está en el backlog.

Una conexión que vive semanas conserva con lo que nació, así que una duración máxima hace que el pool reemplace sus conexiones con regularidad. Debería ser más corta que la duración de la credencial, con alguna variación entre conexiones para que no se reconecten todas en el mismo minuto. El Capítulo 4 explica por qué una credencial vencida no cierra una sesión que ya está abierta; la duración máxima es la forma en que el pool cierra las suyas.

Valores por defecto que una sesión puede levantar

Un timeout fijado como valor por defecto del rol es un valor por defecto. En muchos motores una sesión puede cambiar sus propios ajustes, incluido su timeout de sentencia y su modo de solo lectura, y puede cambiar de rol. Para un servicio eso es aceptable: los límites existen para los accidentes, y el código no levanta su propio timeout por accidente. Para una persona con un cliente SQL no lo es, porque la sesión que necesita el límite es la sesión que puede levantarlo.

Dos controles se sitúan donde la sesión no puede llegar. El gateway abre cada sesión humana con sus límites aplicados y rechaza una solicitud de cambiar los ajustes de una lista protegida; el Capítulo 9 cubre cómo lo hace el gateway. El costo es que los clientes que envían esos ajustes al conectarse dejan de funcionar hasta que alguien los ha listado. Las sesiones humanas obtienen sus límites del gateway: este fija un límite de conexiones por login según el tipo de sesión cuando crea el login de la sesión, y cuenta cuántas sesiones puede mantener una persona. Ninguno es un límite de rol, ya que `human_jit_write` es un techo del que ningún login es miembro. Ninguno se deja en manos del cliente. Para cualquier límite que no tenga ninguno de los dos, la tabla de controles debería decir que es un valor por defecto y no una garantía.

Nombrar a quien llama

Cuando Robin mira las conexiones, la pregunta útil es “¿de quién es esta?”, y el login responde a la primera mitad. Un rol dice que una conexión pertenece a `billing-api`. No dice qué instancia, qué release ni qué ejecución de un job programado. La mayoría de los motores y poolers per-

miten que un cliente adjunte una etiqueta corta a su conexión, normalmente llamada nombre de aplicación, y aparece en el log de la base de datos y en la vista de la actividad actual.

Una convención vale más que la función. Pondría tres cosas en la etiqueta: el servicio, la instancia o ejecución y el release. Por ejemplo, `reporting-job/run-0412/v31` dice qué job, la ejecución de qué noche y qué build. La etiqueta convierte “alguien tiene cuarenta conexiones” en “la ejecución del martes del informe tiene cuarenta conexiones, y es el build nuevo”.

Vienen con dos advertencias. Primera, la etiqueta la proporciona el cliente y puede fijarse a cualquier valor, así que sirve para el diagnóstico y la atribución, no para la autorización. El login es la identidad; la etiqueta es una nota en el sobre. Segunda, un pooler en modo transaction comparte conexiones de servidor entre clientes, así que la etiqueta del lado del servidor puede no ser la del cliente. Algunos poolers la reenvían en cada transacción, otros no, y entonces el log propio del pooler sobre sus clientes es el registro que lleva el nombre. Averigua qué hace tu configuración antes de depender de ello.

La etiqueta también alimenta el Capítulo 14: un registro de auditoría es más fácil de leer cuando cada línea lleva el servicio y la ejecución, no solo un login. El costo es una línea en el arranque de cada servicio, y una convención más que mantener cuando aparece un servicio nuevo.

Las conexiones no son el único recurso que puede consumir un rol. La memoria para ordenar y unir, el espacio temporal en disco, la CPU y el número de workers paralelos también se comparten, y los motores difieren mucho en si pueden limitarlos por rol. Donde el motor no puede, el aislamiento tiene que venir de la ubicación: un lector pesado va a una réplica, que es el argumento del Capítulo 6. Comprueba qué límites puede fijar tu motor por rol antes de prometer alguno a nadie.

Ledgerly: la noche en que el informe se comió la API

Volvamos a la noche del martes. El estado inicial lo explica en una frase: `reporting-job` se ejecutaba como `admin` contra el primary, así que competía por los mismos slots que `billing-api` bajo el mismo nombre, sin límite en ninguno de los dos y sin timeout en nada.

El plan de Alex no empieza por el pooler. Empieza por lo más barato y que más protección da, y avanza por la lista a medida que el equipo aprende.

1. **Límites por rol.** Con los roles de aplicación del Capítulo 3 ya en su sitio, cada uno recibe un límite de conexiones de una tabla como la anterior. El informe pierde de inmediato la capacidad de quitarle los slots a la API, aunque la primera noche siga leyendo el primary.
2. **Timeouts por rol.** `app_billing_api` recibe un timeout de sentencia corto y un límite corto de inactividad en transacción, y el informe recibe un timeout de sentencia largo que no se aplica a la API.
3. **Etiquetas.** Cada servicio y cada ejecución programada fija su etiqueta. El siguiente page nombra la ejecución, no solo el rol.
4. **Un pooler.** Los cinco servicios se conectan a través de él y pasa a ser su única vía de entrada, con el total por rol tomado de uno de los tres lugares anteriores.
5. **Un traslado a la réplica.** `reporting-job` lee de `ledgerly_replica_2` con su propio presupuesto, y el primary deja de cargar con su trabajo. El Capítulo 6 describe el traslado en sí.
6. **Una alerta antes del rechazo.** El equipo de plataforma vigila la saturación de cada pool, no solo los rechazos. Un pool que se mantiene durante horas en, digamos, el noventa por ciento de su límite dice “sube el límite o arregla a quien llama” mientras todavía hay tiempo.

El job también necesita un cambio propio, porque un límite que rechaza al job solo sirve si el job reacciona bien al rechazo. Un job que reintenta de inmediato tras un rechazo es la multitud en la puerta en una forma más pequeña.

```
function acquireWithBackoff(role):  
  for attempt in 1..maxAttempts:  
    try: return acquireConnection(role, waitUpTo: acquireWait)  
    on PoolEmpty: sleep(backoffWithJitter(attempt)) // slow down,  
do not retry harder  
  raise ServiceBusy
```

Con todo esto en su sitio, el mismo cambio que subió el paralelismo del informe tendría otro resultado. El job llega a su límite y es rechazado, se ralentiza a sí mismo y tarda más, y la API no se queda sin slots por su causa. Un job que renuncia a un lote también debería dejar constancia de lo que

omitió, para que nada lea agregados a medio construir como si estuvieran completos; eso es diseño del propio job y va más allá de este capítulo. La revisión que aprobó el cambio habría seguido siendo correcta, porque el diseño ya no depende de que quien revisa advierta la interacción entre dos servicios.

Trade-offs y costos

Los límites por rol cuestan ajuste. Un límite demasiado bajo rechaza carga legítima, y el rechazo parece una caída para el servicio que lo recibe. Los límites necesitan un responsable y una revisión cada vez que un servicio escala o aparece una carga de trabajo nueva, porque la suma de instancias por tamaños de pool puede superar el límite sin que ningún cambio individual parezca incorrecto. El equipo de plataforma pasa a ser el lugar donde se hacen esas preguntas, lo cual es útil y también es una cola nueva.

Un pooler es un sistema nuevo en la ruta. Ahora toda consulta pasa por él, y cuando está caído, todos los servicios están caídos. Debe estar disponible durante el incidente en el que más lo necesitas, así que necesita su propia redundancia, monitoreo y runbook. En modo transaction además elimina funciones de sesión de las que depende algún código, y hace más difícil de leer el log de la base de datos. Si tienes un solo servicio pequeño con una instancia y un pool de diez, bastan el pool propio de la aplicación y un límite de rol.

Los timeouts cancelan trabajo legítimo. Un timeout de sentencia demasiado ajustado mata una buena sentencia, y un retry ingenuo repite el trabajo y la cancelación. Los servicios necesitan manejar la cancelación, y para las escrituras esto significa pensar si la operación puede ejecutarse dos veces sin daño. Los valores de `ledgerly_migrator` y de las correcciones de datos (Capítulo 10) difieren de los de las aplicaciones, lo que supone más configuración que mantener correcta.

Un límite que nadie vigila no protege nada. Un timeout de sentencia tan largo que nunca salta, o un límite de rol que ninguna alerta cubre, añade una línea a la configuración y ninguna cota. La prueba es si alguien lo notaría si el control dejara de funcionar.

Las etiquetas son baratas y falsificables. Ayudan a las personas y cuestan disciplina, pero no prueban nada, y un log que depende de ellas para la atribución depende de una convención.

Cuándo no usar estos controles: no añadas límites por rol antes de que existan los roles, ya que un límite sobre el rol compartido `admin` solo limita a todos a la vez. No fijas timeouts agresivos en un servicio antes de saber cuánto duran sus sentencias normales, porque la primera ejecución de facturas cancelada enseñará al equipo a desactivar los timeouts. Mide primero lo normal y luego fija el límite por encima.

Errores comunes

- **Subir el límite del servidor para resolver un incidente.** Es tentador porque funciona de inmediato y no cuesta nada hoy. Traslada el fallo a una multitud mayor y elimina la señal temprana. Da a cada rol un presupuesto y deja el límite del servidor como la suma.
- **Dimensionar el pool para el tráfico que esperas tener.** Más conexiones parecen más capacidad, pero la base de datos tiene una cantidad fija de concurrencia útil. Dimensiona el pool según el trabajo que una instancia puede hacer a la vez.
- **Asignar todos los clientes a un único login en el pooler.** Es la configuración fácil, y trae de vuelta la cuenta de administrador compartida. Mantén el login propio de cada instancia, o elige explícitamente un login estable por rol y acepta lo que el log ya no dice.
- **Usar el modo transaction con estado de sesión.** El código funcionaba en modo session, así que nadie lo prueba en modo transaction. El contexto de las reglas de fila, la identidad para la auditoría y las tablas temporales deben vivir dentro de la transacción, o el modo debe seguir a nivel de sesión para ese rol.
- **Un único conjunto de timeouts para todos los roles.** Un único valor por defecto es demasiado ajustado para el batch y demasiado laxo para la API, y el resultado habitual es que se sube para todos. Asocia los valores al rol.
- **Tratar el nombre de aplicación como identidad.** La etiqueta la fija el cliente. Usa el login para autenticar y la etiqueta para explicar.

- **Llenar el servidor con presupuestos de aplicaciones.** Cuando los límites suman todo el servidor, el equipo de plataforma no puede conectarse durante el incidente. Mantén una reserva fuera de cada rol de aplicación.

Ideas clave

- Una conexión es un recurso limitado y un punto de política: un límite que pertenece al servidor y no a un rol deja que el primer cliente voraz se lleve la parte de todos.
- Un presupuesto de conexiones por rol convierte “la base de datos está llena para todos” en “este cliente se ha quedado sin slots”, lo que reduce el radio de impacto de un job descontrolado al propio job.
- Mantén una reserva de conexiones fuera de cada rol de aplicación, para que las personas y herramientas que reparan el sistema puedan conectarse siempre.
- El pooler es por donde entran los servicios, así que es el lugar para el rechazo central y el punto de estrangulamiento de red. Muchos poolers no pueden sumar los logins dinámicos de un rol, así que el total por rol sale de la base de datos, del broker o de un login aguas arriba por rol, cada uno con un precio señalado, como expone la sección del pooler.
- El pooling en modo transaction elimina el estado de sesión, así que cualquier contexto del que dependa una regla de fila o un registro de auditoría debe fijarse dentro de la transacción y fallar de forma ruidosa cuando falte.
- Los timeouts de sentencia, de espera de bloqueo y de inactividad en transacción pertenecen al rol, y el timeout de sentencia de la base de datos debería saltar antes que el deadline de la aplicación. Una sesión puede levantar un valor por defecto, así que para las personas el límite debe aplicarse donde no puedan cambiarlo.
- Los nombres de aplicación hacen explicables las conexiones pero no prueban nada, así que el login sigue siendo la identidad y la etiqueta sigue siendo una nota.

Un límite de conexiones y un pooler deciden cuántos clientes llegan al primary, pero no cambian cuánto trabajo cae sobre él. El Capítulo 6 da el siguiente paso: trasladar las lecturas a réplicas y otras copias, y las cuestiones de acceso que llegan con cada copia.

Capítulo 6 — Lecturas, réplicas y copias de datos

Imagina una mañana de martes en Ledgerly, a pocos días del cierre de mes. A Jordan, del equipo de datos, le han pedido una cifra: cuántas facturas se emitieron el mes pasado, divididas por segmento de cliente y por si ya había fallado un intento de pago. El lugar natural para preguntarlo sería la copia de analítica, pero es una restauración nocturna y la restauración tiene un día de antigüedad. La pregunta necesita los datos de hoy. Jordan abre un cliente de escritorio, se conecta con el login de administración compartido y ejecuta una consulta que une `invoices`, `invoice_lines` y `payments` a lo largo de todo el mes.

Nada en la consulta es incorrecto. Es una pregunta razonable, escrita por alguien que hace bien su trabajo. Pero la conexión apunta al `primary`, porque ese es el host que el cliente recordaba, y la consulta lee varias tablas grandes mientras `billing-api` atiende a clientes que están a punto de pagar sus facturas. La aplicación web se ralentiza, Robin recibe un `page`, y Robin y Jordan pasan veinte minutos descubriendo que el informe y la lentitud son el mismo problema. Nadie hizo nada fuera de lo normal. El diseño ofrecía un único lugar desde donde leer, y todas las lecturas fueron allí.

Este capítulo añade un paso del estado objetivo: las lecturas van a réplicas y los informes nunca tocan el `primary`. En el modelo de cinco partes del Capítulo 1, trata del Grant y la Ruta trabajando juntos, que es lo que significa autorización en este libro. El grant decide qué ve quien lee, y la ruta decide a qué base de datos puede llegar siquiera. También trata lo que se sigue de este paso: cada réplica, y cada otra copia de los datos, es un lugar nuevo en el que hay que diseñar el acceso. La pregunta es esta: cuando los datos se leen en más lugares de los que se escriben, ¿qué lecturas van adónde, quién puede leer cada copia y cómo evitas que las copias se multipliquen hasta convertirse en puertas que nadie posee?

El problema y por qué es difícil

Una base de datos que recibe escrituras y además atiende todas las lecturas está haciendo dos trabajos con un único conjunto de recursos. Las escrituras necesitan el primary: solo hay uno, es la fuente de verdad y debe seguir disponible para ellas. Las lecturas se pueden repetir, se pueden repartir, y muchas toleran datos con unos segundos de antigüedad. Sacar las lecturas del primary protege las escrituras, y proteger las escrituras es proteger el libro mayor.

El enfoque obvio es añadir una réplica y apuntar hacia ella las cargas de trabajo con muchas lecturas. Eso funciona durante una semana más o menos. Después aparecen tres problemas, y ninguno lo resuelve la réplica por sí sola.

El primero es que una réplica no es el primary. En la mayoría de los despliegues la replicación es asíncrona: el primary confirma un cambio y la réplica lo aplica poco después. A ese intervalo se le llama **retraso de réplica**. Una lectura que cae en una réplica retrasada devuelve una respuesta verdadera sobre un mundo ligeramente anterior. Para un informe mensual eso es invisible. Para un cliente que acaba de pagar una factura y mira una página que todavía dice “pendiente”, es un ticket de soporte.

El segundo problema es que los privilegios viajan con los datos. Una réplica física es una copia completa, columnas Restricted incluidas, y cualquiera que pueda leer la réplica puede leer lo que podría leer quien lee el primary, salvo que los grants digan otra cosa. La sección siguiente explica por qué un rol no se puede volver más seguro en una sola réplica.

El tercer problema es que las copias se multiplican. La primera réplica es para la aplicación. La segunda es para informes, después para investigación humana, después para el paso de export. Sigue un data warehouse, alimentado por una planificación o por change data capture. Alguien restaura un snapshot en una base de datos de staging. Cada copia fue una respuesta sensata a un problema real de carga o de comodidad en su día. Cada una también contiene datos de producción, tiene sus propias credenciales y su propia ruta de red, y rara vez figura en la lista de vías de entrada a los datos de alguien. Una **puerta lateral** es una copia de datos o una

ruta de export que esquivas los controles de acceso del primary. Este capítulo trata de las copias que construyes a propósito, y el Capítulo 13 vuelve sobre las que aparecen por accidente.

Una política que diga “no ejecutes informes en el primary” no ayuda a Jordan. El cliente recuerda un solo host, el login funciona en todos y el primary es el que siempre tiene datos frescos. El control tiene que estar en cómo se enrutan las lecturas y en a qué es capaz de conectarse cada login.

Dividir la ruta de lectura

Empezaría por tratar la ruta de lectura como una superficie de acceso propia, con su propio inventario, sus propios roles y sus propias reglas, y por hacerle una pregunta a cada lectura: ¿necesita el primary? Lo necesita cuando decide algo, como un saldo antes de cobrar un pago, o cuando debe ver una escritura hecha un momento antes en la misma sesión. Todo lo demás puede ir a una réplica. Cómo clasifica y enruta un servicio sus propias lecturas es diseño de aplicación, y este libro lo deja de lado.

La regla de acceso es la otra mitad, y son dos líneas. Las lecturas de informes y las lecturas humanas nunca recurren al primary como alternativa: si la réplica no está disponible, el job espera o falla y se avisa al equipo de plataforma. Y los logins que se usan para ellas no pueden conectarse al primary en absoluto, de modo que un error no pueda enviarlas allí. Un agregado nocturno que termina con dos horas de retraso es una molestia. Uno que cambia en silencio al primary porque la réplica estaba caída es la forma en que se llegó al estado inicial.

Un rol de solo lectura no es un rol inofensivo

Las lecturas suelen tratarse como carga. También son la principal exposición de confidencialidad: un conjunto de resultados que sale del sistema no cambia nada, no deja nada que revertir y se puede copiar a cualquier parte. Un rol que no puede escribir aún puede leer todas las columnas que tiene concedidas, y una consulta pesada suya puede mantener al primary bajo presión. “Solo lectura” además significa tres mecanismos distintos, que se diferencian en si la sesión puede deshacerlos.

Mecanismo	Qué detiene	¿Puede la sesión deshacerlo?
Un grant sin privilegios de escritura	Las escrituras de ese rol	No por sí mismo; pero una sesión que puede cambiar a un rol con privilegios de escritura ya no queda sujeta a él
Una transacción de solo lectura por defecto	Las escrituras en esa sesión, por defecto	En muchos motores, sí: la sesión puede revertir el ajuste o iniciar una transacción de lectura y escritura
Una réplica física	Toda escritura en ese host	No: el host rechaza las escrituras y la sesión no tiene ningún ajuste para cambiarlo

Para las personas, la réplica es la más fuerte de las tres, y esa es una de las razones por las que las lecturas humanas rutinarias van allí. Lo que ninguna limita es cuánto puede leer una sesión. Un row cap en la sesión y la gradación de los exports pertenecen a la Ejecución, y el Capítulo 9 los cubre.

Los grants viajan a la réplica

La pregunta sobre réplicas que primero hacen quienes revisan la seguridad es “¿qué se puede hacer en ella?”. En los motores que replican físicamente, los roles y los grants viven en el catálogo del clúster, y ese catálogo se copia junto con los datos. Un rol se define una vez, en el primary, y aparece en todas las réplicas. Una réplica es de solo lectura, así que en ella no se puede crear, alterar ni conceder un rol por separado. De ahí se sigue que un rol no se puede volver más seguro en una sola réplica: para acotar la superficie de un lector en una réplica hace falta la capa de conexión o una copia aparte. Los privilegios que diseñaste en el Capítulo 3 están en vigor en todas las réplicas, y a qué base de datos puede llegar un login es una regla de la capa de conexión.

Eso tiene un lado útil. `human_readonly` retiene las columnas Restricted y enmascara las Confidential donde lo indica el Capítulo 12, y lo hace en la réplica por la misma razón que en el primary. `app_reporting_job` lee en una réplica sin acceso a columnas Restricted. Si los grants se construyeron pensando en el mínimo privilegio, una réplica no da a quien lee más de lo que el rol permite.

También tiene un lado arriesgado. En el estado inicial las réplicas aceptan el login de administración compartido, que es propietario de todas las tablas. Quien lee en una réplica con ese login ve todas las columnas, que es la misma exposición que en el primary menos la capacidad de escribir. La gente tiende a considerar seguras las réplicas precisamente por eso, y el resultado es que las réplicas son donde vive el acceso más amplio.

Apuntar cada login a su réplica

El estado objetivo da a cada réplica un propósito, y ese propósito se aplica donde se aceptan las conexiones. De aquí en adelante se usan dos términos en sentidos fijos. Una **política de red** dice qué hosts pueden abrir una conexión TCP a un host de base de datos; el Capítulo 7 escribe una. Una **política de conexiones** dice qué login puede autenticarse en qué base de datos, y desde dónde.

La política de conexiones necesita un mecanismo, porque los grants no pueden llevarla y la política de red no ve los logins. En los motores que lo tienen, el mecanismo es la regla de acceso basada en host que cada servidor lee de su propia configuración. Esa regla se define por el origen de la conexión y por un patrón de login o un rol, y no forma parte de los datos replicados, de modo que el primary y cada réplica pueden tener reglas distintas. El bloque siguiente es una de esas políticas, escrita como las reglas que tendrían los tres servidores.

```
connection policy: one rule set per server, not replicated
ledgerly_replica_1:
    allow from pooler hosts      login app_billing_api,
app_admin_console // browsing, support lookups
    deny everything else

ledgerly_replica_2:
    allow from pooler hosts      login app_reporting_job,
app_vendor_export
    allow from gateway hosts     login pattern: kind readonly //
human_readonly, Chapter 7
    deny everything else

ledgerly_primary:
    deny from anywhere           login app_reporting_job,
app_vendor_export
    allow from gateway hosts     login pattern: kind jit, kind
breakglass
    deny from gateway hosts     login pattern: kind readonly //
routine reads use replica_2
```

El costo es que ahora hay que mantener tres conjuntos de reglas alineados con el esquema de roles, y un cambio que llega a un servidor y no a otro es drift. Los conjuntos de reglas pertenecen al mismo pipeline que aplica las definiciones de roles (Capítulo 3), y una comprobación que compare las reglas de cada servidor con la política pertenece a la misma revisión. Donde un motor no tiene una regla así, las mismas decisiones recaen en el pooler y el gateway, y en la política de red que los admite.

El emparejamiento de `human_readonly` con `ledgerly_replica_2` significa que una persona que investiga un problema lee de la misma réplica que usan los informes. Es un trade-off deliberado: una réplica que proteger, monitorear y dimensionar, a cambio de que la investigación humana compita con el job nocturno por la capacidad. Si ambos empiezan a interferir, una tercera réplica para las personas es una división sensata, y la política de conexiones cambia en un solo bloque.

Las denegaciones en el primary importan tanto como los permisos en las réplicas. `app_reporting_job` y `app_vendor_export` no tienen nada que hacer en el primary. Si sus logins no pueden conectarse allí, el comportamiento de “a veces lee el primary” del estado inicial ya no es algo que un error de configuración pueda hacer volver. Un job apuntado al host equivocado falla al conectarse, que es un fallo visible y barato.

La excepción en el otro sentido es una sesión JIT o break-glass. Todo login de sesión humana es miembro de `human_readonly` por defecto, así que la regla del primary no puede escribirse contra ese rol: o dejaría sin acceso al break-glass o no significaría nada. Se escribe contra el tipo de sesión que abrió el access gateway. Una sesión rutinaria se envía a `ledgerly_replica_2`. Una sesión que necesita escribir, o leer en una emergencia lo que una réplica no puede mostrar con la rapidez suficiente, se abre en el primary, y la solicitud de acceso se hace desde esa sesión (los Capítulos 7, 8 y 10 describen el modelo).

Eso deja una ventana en el primary: hasta que se aprueba una solicitud, una sesión así puede hacer lo que hace `human_readonly`. Tres reglas mantienen pequeña la ventana, y el Capítulo 7 las enuncia como parte de los tipos de sesión. Solo pueden abrirla las personas a las que la política permite solicitar ese tipo. El motivo se registra cuando la sesión se abre, y la sesión tiene una ventana corta de apertura en la que debe llegar una soli-

cidad. Si no llega ninguna, salta una alerta. Quien solo quiere mirar datos no abre una sesión así; para eso está la réplica, y por eso las lecturas rutinarias no se permiten en el primary.

La réplica también tiene un host

Una réplica física es una copia de los archivos de datos, así que sus hosts, volúmenes, snapshots y credenciales de replicación llegan a los mismos datos sin pasar por un rol de base de datos. Aplicaría a las réplicas las mismas reglas de acceso a hosts, el mismo cifrado de almacenamiento y la misma retención que al primary, y lo dejaría escrito al crear la réplica, porque nadie se acordará después. El Capítulo 13 vuelve sobre los snapshots y los backups.

Informes, analítica y la elección de la copia

Las réplicas resuelven un problema de carga para lecturas baratas y frecuentes. Los informes son lo contrario: lecturas raras, grandes y lentas. La pregunta es si los informes pertenecen a una réplica o a otra cosa, y la respuesta depende de lo que necesiten los informes.

Tres lugares desde donde puede leer un informe

Opción	Frescura	Carga en el primary	Superficie de acceso	Encaja con
Una réplica de informes (ledgerly_replica_2)	De segundos a minutos	Solo la replicación	Los mismos datos y roles que el primary	Informes operativos, exports, investigación
Una copia aparte cargada con una planificación (ledgerly_analytics)	Tan antigua como la última carga	Una carga por planificación	Sus propios roles y sus propios datos	Modelos, dashboards, análisis masivo del equipo de datos
Una copia alimentada por change data capture	De segundos a minutos	Un lector del flujo de cambios	Un pipeline nuevo y un almacén nuevo	Analítica casi en tiempo real, muchos consumidores

Enviaría los informes operativos a una réplica y el análisis masivo a una copia aparte, y me resistiría a poner el trabajo del equipo de datos en la réplica por una razón que no tiene que ver con la carga. Una réplica lleva la forma y la sensibilidad del esquema de producción. El equipo de datos no necesita las columnas Restricted, y en el estado objetivo tampoco ve las columnas Confidential en bruto, porque la copia de analítica se enmascara al cargarse, salvo las pistas de visualización que existen para mostrarse (el Capítulo 12 describe cómo y nombra la excepción). Una réplica no se puede enmascarar por consumidor sin cambiar lo que la réplica es. Una copia aparte se puede construir para su público.

Por eso mismo la copia de analítica no se llama réplica en ninguna parte de este libro. Se carga con una planificación desde el primary, tiene roles distintos y su contenido puede diferir del origen. Llamarla réplica invita a suponer que lleva los mismos controles, y en el estado inicial lleva menos.

Change data capture es la opción a la que recurriría en último lugar. Añade un pipeline que hay que mantener en marcha, un almacén que hay que asegurar, un retraso propio y un contrato de esquema entre productor y consumidor. Para una empresa del tamaño de Ledgerly tomaría primero la carga planificada y dejaría por escrito qué me haría cambiar de opinión: por ejemplo, informes que necesiten datos más recientes de lo que permite la planificación, o un segundo equipo con la misma necesidad.

El costo de cada copia

Todas las opciones de la tabla tienen el mismo elemento oculto: una cosa nueva que proteger. Una copia es un lugar donde viven los datos, un conjunto de logins que pueden leerlos, una ruta de red, una regla de retención y un responsable. Una copia sin responsable es una puerta lateral, y una copia con responsable y sin vencimiento es una puerta lateral que tarda más en notarse.

Ayuda llevar un registro breve de copias, con los mismos campos para todas. El bloque siguiente es notación ilustrativa, y lo importante es que las líneas `owner` y `ends` no pueden quedar en blanco.

```

table data_copy:
  name          text primary key    // "ledgerly_replica_2"
  kind          text not null       // replica, analytics, export,
dump, staging
  source        text not null       // what it was made from
  contains      text not null       // data classes present:
Restricted, Confidential, ...
  readers       text not null       // roles or logins that may
read it
  owner         text not null       // a team, never a person
  ends         timestamp null      // null only for replicas and
the analytics copy
  review_due    timestamp not null

```

En Ledgerly el registro empieza con cinco entradas: las dos réplicas, la copia de analítica, los archivos de export que van al proveedor de contabilidad y los dumps antiguos en el almacenamiento compartido. Cuatro son deliberadas y los dumps no, que es lo que hace útil el registro: los dumps reciben un responsable y una fecha de fin cercana. El Capítulo 13 usa el mismo registro para las copias que viven fuera de las bases de datos.

Nota: Un registro que nadie abre no es un control, así que se ata al paso que crea las copias. El paso de aprovisionamiento se niega a crear una réplica, una restauración de snapshot, una base de datos de staging o un job de export a menos que exista una fila con un responsable y, donde el tipo lo exija, una fecha de fin. La revisión de accesos (Capítulo 14) pregunta entonces qué copias existen y quién puede leer cada una.

La negativa tiene un costo: alguien rellena una fila antes de que un equipo con una fecha límite pueda obtener su copia, y las primeras semanas generan quejas. Además cubre solo las copias hechas mediante el paso de aprovisionamiento, de modo que una copia hecha a mano sigue siendo una puerta lateral, y el Capítulo 13 se ocupa de ella.

Alcance, no solo grants

Un rol correcto sobre el papel sigue siendo incorrecto si puede conectarse el login equivocado. Para cada copia, el grant retiene lo que el público no necesita (Capítulos 3 y 12), y el alcance decide qué rutas de red y qué principals pueden abrir siquiera una conexión. En una red plana, un login

filtrado de cualquier servicio llega a todas las copias, y el alcance acota lo que vale una filtración, como dice el Capítulo 4. La tabla enuncia la regla de alcance para cada lugar; la política de red que hace cumplir el lado del host es la del Capítulo 7, y la política de conexiones de arriba hace cumplir el lado del login.

Lugar	Accesible desde	No debe ser accesible desde
ledgerly_primary	Los hosts del pooler, el gateway para sesiones JIT y break-glass, el paso del pipeline que aplica las migraciones	Los jobs de informes y de export, las sesiones humanas rutinarias, la capa web directamente
ledgerly_replica_1	Los hosts del pooler, para los servicios que navegan	El gateway para lecturas rutinarias, la carga de analítica, la capa web directamente
ledgerly_replica_2	Los hosts del pooler para informes y export, el gateway para las lecturas humanas rutinarias	Los servicios que ejecutan la aplicación, la capa web directamente
ledgerly_analytics	El job de carga, reporting-job para sus resultados, el gateway para sesiones de analítica	Los servicios que ejecutan la aplicación

Migrar la ruta de lectura de Ledgerly

Considera cómo movería Alex a Ledgerly del estado inicial a este diseño sin romper el job nocturno al primer intento. El orden importa más que la velocidad, y yo lo haría en cinco pasos.

- 1. Inventariar las lecturas.** Para cada servicio, lista las consultas que ejecuta y clasifícalas en lecturas que necesitan el primary y lecturas que no. La mayoría de las consultas de listado de `billing-api` son del segundo tipo. Sus escrituras y sus redirecciones posteriores a una escritura no lo son. Los logs de consultas son la materia prima de las reglas de enrutamiento.
- 2. Dar a cada réplica un propósito.** Escribe la política de conexiones de `ledgerly_replica_1` y `ledgerly_replica_2` y aplícala primero a los logins nuevos, dejando por el momento el login compartido en su sitio.

3. **Mover reporting-job y el paso de export.** Apunta ambos a `ledgerly_replica_2` con sus propios logins y después deniega esos logins en el primary. Ejecútalos en paralelo durante unos ciclos y compara las salidas antes de apagar la ruta antigua.
4. **Mover las lecturas de navegación.** Enruta las lecturas de navegación de `billing-api` y `admin-console` a `ledgerly_replica_1` detrás de un flag, y sube la proporción poco a poco.
5. **Cerrar el login compartido en las réplicas.** Cuando nada legítimo lo use allí, quítale la capacidad de conectarse a cualquiera de las dos réplicas. Este es el paso que hace permanente todo lo demás, y es el que los equipos posponen.

La ruta del equipo de datos también cambia. La pregunta de cierre de mes de Jordan ahora tiene dos respuestas. Si una cifra de hace un día sirve, va a `ledgerly_analytics` mediante `human_analytics_reader`. Si necesita los datos de hoy, va a `ledgerly_replica_2` a través del access gateway con `human_readonly`, una sesión que queda registrada, que no puede ver columnas Restricted y que no puede tocar el primary. Lo que Jordan pierde es la posibilidad de apuntar un cliente de escritorio a cualquier host y que funcione. Lo que gana es que una consulta pesada sea un problema para la réplica y uno leve para todos los demás.

Proteger una réplica de quienes la leen

Una réplica que atiende a personas y a un job nocturno recibirá consultas pesadas, y en algunos motores una consulta larga puede retrasar la replicación o cancelarse para que la replicación continúe, según la configuración. Los controles son los del Capítulo 5, aplicados a los roles de la réplica: un statement timeout más corto para `human_readonly` que para `app_reporting_job`, un límite de conexiones por login que el gateway fija en cada login de sesión humana, y un paso de export que se ejecute fuera de la ventana del agregado nocturno. Añade una alerta de retraso y envía el page a quien figure en el registro como responsable de la réplica. Una alerta que nadie recibe no es un control.

Trade-offs y costos

Una réplica es un sistema que operar. Hay que monitorearla, parchearla, dimensionarla y recuperarla, y para un equipo pequeño una réplica y una copia nocturna pueden ser todo lo que hace falta. El argumento de acceso sigue en pie sin ella: tú decides qué logins llegan al primary y cuáles no.

Compartir `ledgerly_replica_2` tiene un precio. Las personas y el job nocturno compiten por la capacidad de una sola réplica, a cambio de un único lugar que proteger. La división en una tercera réplica es barata en términos de política y cara en operaciones.

La política de conexiones es más configuración. Las reglas por servidor tienen que coincidir con el esquema de roles y sobrevivir a un failover, a una reconstrucción y a una réplica nueva. El drift entre servidores es el fallo, y la comprobación comparativa es el precio de mantenerlo visible.

Algunas pantallas irán retrasadas. Sacar las lecturas del primary implica retraso, y alguien lo notará. Acuerda de antemano con soporte y producto qué pantallas pueden ir retrasadas, y comprueba primero el retraso cuando algo no cuadre en una investigación sobre una réplica (Capítulo 15).

El registro añade fricción. Exigir una fila antes de que exista una copia frena al equipo que la quería, y esa lentitud es el efecto buscado. Un registro que nadie tiene que rellenar al crear es una lista de lo que alguien recordó.

Hay casos en los que el consejo de este capítulo no se aplica. Un sistema pequeño cuyo primary tiene margen y donde las ingenieros que leen de él ya se conectan por una ruta controlada puede no necesitar una réplica por carga, y separar la ruta de lectura rinde menos. Un sistema con requisitos de frescura muy estrictos para todas las lecturas puede tener que ejecutar la mayoría en el primary y poner su esfuerzo en limitar lo que esos lectores pueden hacer. Una réplica tampoco es un backup; el Capítulo 13 trata los backups.

Errores comunes

- **Tratar una réplica como menos sensible que el primary.** No se puede escribir en ella, así que acumula acceso amplio: el login comparti-

do, un rol de lectura permanente para todos, una regla de firewall más ancha. Contiene los mismos datos, así que recibe los mismos grants y una política de conexiones igual de estrecha.

- **Dejar que los informes o las lecturas humanas recurran al primary.** Recurrir a él es tentador porque mantiene en marcha el informe cuando la réplica está caída. Haz fallar la lectura en su lugar, y deniega esos logins en el primary para que un error no pueda enviarlos allí.
- **Enviar a una réplica lecturas que deciden algo.** La lectura es barata y la réplica está ociosa, así que parece una ganancia fácil. Una decisión tomada sobre datos obsoletos y seguida de una escritura es la forma en que se desajustan los saldos y los estados. Clasifica las lecturas por lo que las sigue, no por lo pesadas que sean.
- **Construir un data warehouse apuntando a una persona a una réplica.** El equipo de datos carga entonces con el esquema de producción, con la sensibilidad de producción y con los límites de capacidad de producción. Dale a la analítica su propia copia y sus propios roles (Capítulos 12 y 13).
- **Añadir copias sin un registro.** Cada una fue un favor a un equipo con una fecha límite. Seis meses después nadie sabe decir cuántas existen. Haz que el responsable y la fecha de fin formen parte de la creación de una copia.
- **Olvidar el host y los snapshots.** Los roles son correctos, y el snapshot del volumen de la réplica lo puede leer cualquiera con acceso a la infraestructura. Incluye el host, el almacenamiento y los snapshots en los controles de cada copia.
- **Suponer que un rol se puede restringir por réplica.** En los motores que replican físicamente, los roles son de todo el clúster. Acota la superficie de un lector en la capa de conexión o con una copia aparte.

Ideas clave

- Las escrituras necesitan el primary y la mayoría de las lecturas no, así que la primera pregunta sobre la ruta de lectura es si una lectura decide algo o sigue a una escritura; las lecturas de informes y las humanas nunca recurren al primary.

- “Solo lectura” son tres mecanismos, y solo la réplica no puede deshacerse desde una sesión, de modo que un rol de solo lectura limita las escrituras y no lo que una sesión puede leer.
- Una réplica contiene los mismos datos que el primary y se rige por los mismos roles de todo el clúster, así que los grants del Capítulo 3 y la política de conexiones deciden juntos lo que puede ver quien lee.
- La política de conexiones es una regla por servidor definida por origen y login, mantenida alineada con el esquema de roles; cada réplica tiene un propósito, y el login de administración compartido no se acepta en ninguna.
- Una sesión JIT o break-glass en el primary necesita una persona que pueda abrirla, un motivo registrado, una ventana corta y una alerta cuando no sigue ninguna solicitud.
- La analítica pertenece a una copia aparte construida para su público, y llamarla réplica oculta que lleva controles distintos.
- Toda copia de datos necesita un responsable, una lista de lectores y, salvo las réplicas y la copia de analítica, una fecha de fin, y el paso que crea copias se niega a ejecutarse sin ellos.

La ruta de lectura ya tiene sus propios roles, sus propios destinos y una lista breve de copias, pero todavía no hemos dicho cómo llega una persona a ninguno de ellos. El próximo capítulo aborda las rutas de entrada para las personas, y compara un bastion, un database proxy y un access gateway.

PARTE III

Acceso humano

Capítulo 7 — Rutas de entrada: bastions, proxies y access gateways

Imagina que a Alex, del equipo de plataforma, le hacen una pregunta sencilla durante una reunión de planificación: “Si mañana apagáramos una ruta de entrada a la base de datos de producción, ¿cuál dolería?”. Nadie responde rápido. Alex va a averiguarlo y empieza por la red. La VPN da a cada ingeniero una ruta hasta el primary. También hay un host bastion que alguien montó para un contratista hace dos años y que todavía tiene un puerto a la escucha. Los clientes de escritorio de una docena de laptops guardan conexiones, y una de ellas se configuró para hacer un túnel a través del bastion, mientras que otra va directa por la VPN.

Después Alex le pregunta a Robin, que esta semana está de on-call, cómo llegaría Robin a la base de datos de noche. Robin nombra dos rutas sin dudar y una tercera después de pensarlo. Cada ruta funciona, ninguna fue diseñada, y cada una termina en el mismo login de administración compartido. El equipo de red puede decir quién está en la VPN. La base de datos puede decir que el login de administración se conectó. Nadie puede decir qué persona estaba detrás de una conexión dada, y nadie puede apagar el acceso de una persona sin apagar el de todas.

Así que la pregunta de este capítulo es: ¿qué debería interponerse entre una persona y la base de datos para que llegar a la red y llegar a los datos sean dos decisiones separadas, cada una con un nombre asociado? Este capítulo añade el access gateway, el primer paso del lado humano del estado objetivo de Ledgerly. En las cinco partes del libro lleva la Ruta, y la identidad de quien entra por ella: la red decide quién puede llamar a la puerta, y el gateway pone un nombre a cada sesión. Los roles que recibe una sesión del gateway, y cómo pide una persona más que lectura, pertenecen al Capítulo 8.

El problema y por qué es difícil

Una ruta de red y una ruta de base de datos parecen lo mismo, porque la primera es lo que necesitas antes de poder intentar la segunda. Una persona que está en la VPN puede abrir una conexión al puerto de la base de datos. Que luego pueda hacer algo lo deciden un nombre de usuario y una contraseña, y en el estado inicial de Ledgerly la contraseña es compartida. La red dijo “puedes llamar a la puerta”, y la base de datos no pudo saber quién llamó.

Ese es el núcleo del problema. Los controles de red responden a la pregunta “qué máquinas pueden hablar con qué máquinas”, y lo hacen bien. No responden a “quién es esta persona, qué se le permite hacer ahora mismo y qué hizo”. Cuando se le pide a una regla de red que sustituya a esas preguntas, falla de manera silenciosa: la regla sigue siendo correcta mientras el acceso que concede queda obsoleto. Una persona ingeniera que cambió de equipo sigue estando en la VPN. Un contratista cuyo contrato terminó sigue en un grupo que nadie revisa.

Otras tres cosas hacen más difícil la ruta humana que la ruta de servicio.

Primero, las personas son los clientes más variados. Un servicio se conecta desde un lugar conocido con un driver conocido. Una persona puede usar un cliente de línea de comandos, una GUI de escritorio, un notebook, una herramienta de navegador o un script que escribió el martes pasado. Cada uno tiene su propia manera de recordar credenciales, su propio valor por defecto de auto-commit y su propio hábito de descargar resultados al disco local. El Capítulo 1 enumeró los clientes que recuerdan la contraseña y hacen auto-commit; no es una configuración inusual, es el comportamiento por defecto de muchas herramientas hechas por comodidad.

Segundo, las personas se conectan desde los lugares menos controlados. Un laptop viaja, se suspende, se respalda y a veces se comparte en casa. Una conexión guardada con una contraseña dentro se mueve con el laptop, y lo mismo hace cualquier conjunto de resultados exportado a un archivo. La base de datos no ve nada de eso.

Tercero, la ruta se usa bajo presión. La persona que necesita la base de datos de noche no está de humor para aprender una herramienta nueva, y una ruta lenta o frágil la esquivará quien tenga los medios. Un diseño co-

rrecto sobre el papel que pierde frente al viejo atajo a las 3 a. m. no ha cambiado nada, y por eso el atajo hay que eliminarlo y no solo desaconsejarlo.

Una política que diga “no compartas el login de administración” no se puede comprobar, porque el log de la base de datos muestra el login de administración y nada más. Un diseño en el que cada sesión tiene su propio login sí se puede comprobar, y también una regla de red que deje que solo una cosa llegue a la base de datos. El resto de este capítulo compara las maneras de construir un diseño así.

Cuatro rutas de entrada, comparadas

Antes de la comparación, dos términos, porque en la industria las palabras se usan con holgura y el libro necesita fijarlas.

Un **bastion** es un host endurecido que es la única ruta de red hacia un sistema protegido. Reenvía conexiones. No entiende el protocolo de la base de datos, así que no distingue un `SELECT` de un `DROP`, y no sabe quién está detrás de la conexión a menos que la persona haya iniciado sesión en el propio bastion.

Un **access gateway** es un intermediario consciente de la identidad que autentica a una persona, emite una credencial de vida corta y registra la sesión. Un **database proxy** es algo distinto: un intermediario que entiende el protocolo y que agrupa, enruta o filtra conexiones a bases de datos. Un gateway puede construirse sobre un database proxy, pero los dos responden a preguntas distintas. Un proxy conoce el protocolo; un gateway conoce a la persona. Este capítulo usa cada término únicamente en este sentido.

Estas son las rutas que un equipo suele considerar, con lo que cada una sabe de la persona y lo que deja atrás. Las cuatro primeras ponen a una persona en una sesión; las dos últimas la mantienen fuera de ella.

Ruta	Qué sabe de la persona	Qué muestra el log de la base de datos	Costo principal
Endpoint público	Nada más allá del login de la base de datos	Un login, desde cualquier dirección	Expone el puerto de la base de datos a todo internet

Ruta	Qué sabe de la persona	Qué muestra el log de la base de datos	Costo principal
VPN a la base de datos	Que la persona está en la red	El login compartido o por persona	La pertenencia a la red se trata como identidad
Bastion	Quién inició sesión en el host bastion, si las cuentas son por persona	El login que la persona escribió, desde la dirección del bastion	Un host más que parchear, y no ve sentencias
Access gateway	La identidad del inicio de sesión único de la empresa	Un login de sesión que corresponde a una persona y a una sesión	Un sistema nuevo que debe estar disponible cuando alguien lo necesita
Ruta de producto, sin SQL	El rol de la persona en una herramienta interna	El login propio del servicio, con la persona en el rastro de auditoría de la herramienta	Hay que construir una funcionalidad para cada tarea; las preguntas libres no tienen ruta
Ejecutor de enviar y ejecutar	La persona que envió y quien revisó y aprobó	El login del ejecutor, ejecutando un conjunto de sentencias revisado	Cada tarea espera revisión; nada es interactivo

Las dos últimas filas no son maneras de llegar a una base de datos en absoluto. La persona nunca tiene una sesión, y la sentencia se ejecuta desde un componente que la comprobó antes. Pertenecen a la comparación porque para muchas tareas son la mejor respuesta: una persona que nunca abre una sesión no puede dejar una abierta. El Capítulo 1 presentó el abanico de esos patrones, el Capítulo 9 cubre cómo comprueba el gateway una sentencia y el Capítulo 10 cubre los patrones de corrección de datos; este capítulo cubre las sesiones que quedan después de que las tareas que encajan en esos patrones se hayan ido.

El endpoint público

No expondría la base de datos a internet público para uso humano, y el argumento es corto. Un puerto accesible invita a tráfico automatizado que no tiene nada que ver con las personas de Ledgerly, y todas las credenciales de esa base de datos pasan a ser algo que vale la pena adivinar. La única defensa que queda es la fortaleza del login, y la premisa del libro es que los logins se filtran.

Hay casos limitados en los que una base de datos gestionada solo es accesible públicamente y el proveedor ofrece una allowlist de direcciones de origen. Si estás en esa situación, trata la allowlist como el mínimo y pasa a una ruta privada en cuanto el proveedor lo permita. La prueba es simple: ¿podría alguien que nunca ha trabajado para ti abrir una conexión al puerto? Si es así, todos los demás controles de este libro parten de un punto peor.

La VPN

Una VPN es una capa de red razonable. Reduce quién puede llegar a la base de datos a las personas a las que se ha dado una ruta, y la mayoría de las empresas ya tiene una. Su límite es lo que demuestra. Demuestra que una persona, o el laptop de una persona, pasó las comprobaciones de la VPN. No dice qué login de base de datos usó después y no registra sentencias.

En el estado inicial de Ledgerly, la VPN es todo el control. Todas las personas que están en ella pueden llegar al primary, y el login de administración es el mismo para todas. Mantendría la VPN, pero le daría un trabajo más pequeño: lleva a las personas al gateway y a las demás herramientas internas, y ya no las lleva al puerto de la base de datos.

El bastion

Un bastion añade una propiedad útil: un único lugar donde se puede cerrar la puerta de red. El firewall de la base de datos solo permite la dirección del bastion, así que apagar una cuenta del bastion apaga la ruta de red de una persona. Si el bastion tiene cuentas por persona y registra quién inició sesión en él, obtienes un registro aproximado de quién estuvo conectado y cuándo.

Lo que un bastion no da es nada por debajo de la capa de red. Una vez que una persona tiene un túnel abierto a través de él, la conexión a la base de datos es lo que el cliente de esa persona haga de ella. El login lo sigue escribiendo la persona o lo sigue guardando su cliente. Las sentencias son invisibles para el bastion. Así que un bastion ordena el panorama de red sin arreglar el panorama de identidad, y por eso un equipo puede tener uno y aun así no poder responder a “¿quién ejecutó esa sentencia?”.

Un bastion también tiene un problema de mantenimiento silencioso. Es un host con una shell, así que necesita parches, gestión de claves y un responsable que recuerde que existe. El host del contratista del comienzo es la forma habitual de este fallo: era correcto el día en que se construyó y después nadie quedó a cargo de él.

El access gateway

El access gateway es adonde enviaría a las personas, por las razones que siguen.

Autentica a la persona contra el inicio de sesión único de la empresa, de modo que la identidad es la misma que la empresa ya controla. Dejar la empresa, cambiar de equipo y fallar una comprobación multifactor surten efecto en el gateway sin que nadie edite la base de datos. Emite una credencial que pertenece a una sesión y muere con ella, así que no hay contraseña que guardar. Y registra la sesión, de modo que “¿quién ejecutó esa sentencia?” tiene respuesta en dos lugares: el registro de sesión del gateway y el log de la base de datos, que nombra un login que corresponde exactamente a una persona.

Los gateways vienen en dos formas generales. Unos se sitúan en la ruta de datos: el cliente de la persona se conecta al gateway, y el gateway abre la conexión a la base de datos en nombre de la persona. Otros solo intermedian la credencial y se apartan, de modo que el cliente se conecta directamente a la base de datos con el login de vida corta. La primera forma puede registrar sentencias y hacer cumplir reglas sobre la conexión. La segunda es más simple y pone menos cosas en la ruta, pero entonces la red tiene que permitir que las personas lleguen al puerto de la base de datos, lo que devuelve parte del problema con el que empezó este capítulo.

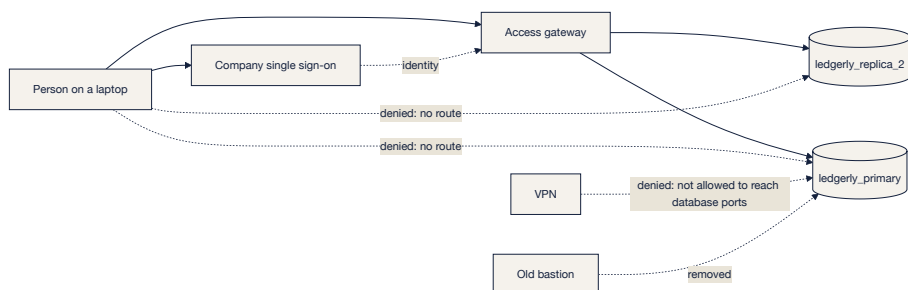
Para Ledgerly elegiría la forma en la ruta de datos, porque permite que la regla de red sea absoluta: solo la dirección del gateway puede llegar a los hosts de la base de datos. El precio es que el gateway maneja cada sentencia humana, así que su capacidad y su disponibilidad importan. Vuelvo sobre ello más abajo.

Separar el alcance de red del acceso a la base de datos

El cambio más útil de este capítulo no es un producto. Es separar dos preguntas que el estado inicial responde juntas.

La primera pregunta es el **alcance de red**: qué máquinas pueden abrir una conexión a qué host y puerto de base de datos. La segunda es el **acceso a la base de datos**: qué login puede hacer qué una vez conectado. En el estado inicial, la VPN responde a la primera y una contraseña compartida responde a la segunda. Después del cambio, una regla de red responde a la primera para todos a la vez, y el gateway responde a la segunda para cada persona por separado.

El diagrama muestra la disposición objetivo para las personas. Fíjate en las flechas denegadas: son las rutas que existen hoy y que no existirán después.



Detrás de ese diagrama hay dos puntos de aplicación. La regla de red de los hosts de la base de datos permite conexiones solo desde el gateway, desde las redes propias de los servicios, desde el pipeline y desde el credential broker. Y el gateway solo reenvía una sesión a la base de datos que permite el tipo de la sesión: una sesión de lectura rutinaria va a `ledgerly_replica_2`, y una sesión abierta para hacer una solicitud JIT o break-glass va a `ledgerly_primary`, como expuso el Capítulo 6.

Escribiría la regla de red como una lista corta que cualquiera pueda leer, porque la lista es lo que comprueba una revisión de accesos. El Capítulo 6 definió la diferencia: esta lista es la política de red, que decide qué hosts pueden conectarse en absoluto, mientras que la política de conexiones decide qué login puede autenticarse en qué base de datos. Si contiene una entrada que nadie sabe explicar, esa entrada es una ruta.

```

network policy (illustrative):
  ledgerly_primary:
    allows  access gateway
    allows  service networks of billing-api, invoice-worker,
           payments-service, admin-console
    allows  deployment pipeline
    allows  credential broker, for access administration
    denies  everything else, including the VPN

  ledgerly_replica_1:
    allows  service networks of the read-only services
    allows  replication traffic from the primary
    denies  everything else, including the VPN

  ledgerly_replica_2:
    allows  access gateway
    allows  reporting-job network
    allows  replication traffic from the primary
    denies  everything else, including the VPN

  ledgerly_analytics:
    allows  analytics-job network
    allows  access gateway, masked views only
    denies  everything else, including the VPN

```

La misma lista continúa con lo que sale de los hosts de la base de datos y con lo que puede llegar al gateway.

```

network policy (illustrative, continued):
  egress from the database hosts:
    allows  replication, the audit log, backups
    denies  everything else

  the gateway's own exposure:
    reachable from the VPN and the office network, nothing else
    reaches       the database hosts, the audit log, single
sign-on

```

Fíjate en lo poco que tiene que hacer la VPN en este modelo. Ya no custodia la base de datos. Lleva a las personas al gateway, y el gateway decide el resto. La VPN importa menos; el acceso de cada persona queda ahora atado a una sesión con nombre.

Dos entradas de la lista son fáciles de olvidar. La primera es el egress. Un host de base de datos que puede abrir conexiones salientes a cualquier parte puede usarse para sacar datos, y una allowlist corta (replicación, el registro de auditoría, backups) convierte eso en un evento visible. La se-

gunda es la propia exposición del gateway: ahora es la única puerta accesible, así que él mismo figura en una lista corta, y no debe ser accesible desde ningún lugar al que la VPN no llegue ya.

La lista también supone que la red puede distinguir a quienes llaman. En una red de clúster compartida, donde varios servicios vienen del mismo rango de direcciones, no puede: una regla que permite “las redes de los servicios” las permite todas. En ese caso la regla de red deja fuera todo lo que no es un servicio ni el gateway, y son el pooler y el login los que separan un servicio de otro (Capítulos 5 y 6). Dejaría eso escrito como la regla del clúster, para que nadie lea la lista de red como más precisa de lo que es. Qué login puede autenticarse en qué base de datos es la política de conexiones del Capítulo 6, y este capítulo solo remite a ella.

El access gateway en Ledgerly

Esto es lo que hace el gateway, en el orden en que lo hace, y lo que ocurre cuando un paso falla. Esta es la parte que probaría con cuidado, porque un gateway que falla abierto se convierte en la nueva contraseña compartida con mejor marca.

Qué es una sesión

Una sesión empieza cuando una persona se autentica. El gateway comprueba la identidad, decide qué tipo de sesión puede solicitar la persona (una sesión de lectura rutinaria para la mayoría; los tipos JIT y break-glass solo mediante las solicitudes del Capítulo 8) y luego crea un **login de sesión**, la cuenta que usa la sesión de la persona. Se crea en `ledgerly_primary` y se replica (Capítulo 6). Después la sesión se conecta a la base de datos a la que pertenece su tipo. Ese login lleva la identidad de la persona en su nombre, de modo que aparece en el log de la base de datos, y es miembro de `human_readonly` por defecto. La credencial se genera para esta sesión, se entrega al gateway y nunca se muestra a la persona, así que no hay nada que guardar.

El nombre de un login tiene una forma fija, para poder leer una línea de log sin consultar nada: un prefijo que dice que es un login de sesión, el identificador corto de la persona y el número propio de la sesión, como en

`sess-robin-0412`. El nombre identifica; el registro del gateway es lo que lo asocia con la persona y la solicitud, y nada debería analizar el nombre para decidir lo que la sesión puede hacer.

Para crear y eliminar logins, y para conceder y revocar pertenencias a roles, el gateway necesita privilegios en la base de datos. Los obtiene mediante su propio login para el rol `ledgerly_access_admin`. El paso del pipeline para las definiciones de roles y el credential broker también usan ese rol, cada uno mediante su propio login (Capítulos 3 y 4). Eso convierte el login del gateway en una credencial sensible: quien la tenga puede conceder cualquier cosa dentro de los roles que administra. La guardaría en el mismo lugar y bajo las mismas reglas que las credenciales del propietario, no le daría ningún privilegio sobre datos y auditaría cada uso (Capítulo 14). Copiarla en scripts o en el chat por comodidad recrearía la contraseña compartida un nivel más arriba.

Quién puede abrir qué tipo de sesión

`mayRequestKind` es política, y yo la mantendría como datos que un revisor pueda leer, no como una condición enterrada en el código del gateway. Una tabla la responde: el grupo del inicio de sesión único, los tipos que ese grupo puede abrir, lo máximo que puede durar cada uno, la base de datos en la que aterriza, sus valores por defecto (solo lectura, statement timeout, row cap) y si puede hacer export.

```
session policy (illustrative):
  group backend-engineers:
    routine      max 8 hours, ledgerly_replica_2, read-only, row
cap, no export
    primary-bound max 2 hours, ledgerly_primary, only to file a
request
  group analysts:
    routine      max 8 hours, ledgerly_analytics, masked views,
export logged
  group contractors:
    routine      max 4 hours, ledgerly_replica_2, masked views,
no export
```

Los tipos dirigidos al primary son los sensibles, ya que una sesión en el primary está a una solicitud de distancia del acceso de escritura. Cuatro reglas los mantienen poco frecuentes: solo los grupos con nombre pueden abrir uno; debe abrirse con un motivo registrado; el gateway espera solo

un tiempo corto, digamos quince minutos, a que se presente una solicitud desde ella; y una sesión que termina sin solicitud genera una alerta, porque “abrió el primary y no hizo nada” es un patrón que vale la pena notar. El tope de dos horas rige solo hasta un grant: cuando `grantJitAccess` o `declareBreakGlass` del Capítulo 8 aprueban una solicitud, el gateway restablece el `ends_at` de esa sesión al `expires_at` de la solicitud, y `endSession`, más abajo, cierra la sesión en ese momento. El Capítulo 8 parte del otro lado, con la solicitud.

El orden de las operaciones

El gateway, el registro de auditoría, la base de datos y el inicio de sesión único son sistemas separados. Ninguna transacción los abarca, así que hay que diseñar el orden de los pasos y el tratamiento de una sesión a medio terminar.

```

function openSession(person, wantedKind):
    identity = verifySingleSignOn(person)
    if not mayRequestKind(identity, wantedKind):
        raise SessionRefused
    login = newLoginName(identity)
    endsAt = now() + maxSessionLength(wantedKind)

    // the record comes first: a login with no record is what the
    sweep hunts
    recordSession({ login: login, identity: identity, kind:
wantedKind,
                    ends_at: endsAt, status: "opening" })

    try:
        auditLogAppend({ actor: identity, action: "session-open",
                        subject: login, ends_at: endsAt })
    on AuditLogUnavailable:
        markSessionRefused(login)
        raise SessionRefused

    try:
        // always on the primary: logins are cluster-wide and
    replicate
        createLogin(login, on: "ledgerly_primary", password:
freshSecret())
        grant("human_readonly", to: login)
        // replication lag: wait until the target database knows the
    login
        waitUntilLoginVisible(login, on: databaseFor(wantedKind))
    on DatabaseError:
        // a login with no role is harmless, but it must not linger
        dropLoginIfExists(login)
        markSessionRefused(login)
        raise SessionRefused
    markSessionOpen(login)
    return connectionFor(login, databaseFor(wantedKind))

```

Lee el bloque por su tratamiento de fallos. El registro se escribe antes de que exista el login, de modo que una caída a mitad deja un registro que dice “opening” y ningún login, o un login que el registro explica. La entrada de auditoría va antes que el login: si el almacén del log es inalcanzable, no se abre ninguna sesión. Esa elección tiene un costo, que la sección de trade-offs nombra. Para una sesión de lectura rutinaria el gateway espera hasta que `ledgerly_replica_2` haya recibido el login, porque de otro modo el retraso de replicación haría fallar la primera conexión de vez en cuando. Si falla crear el login o conceder el rol, el gateway intenta limpiar y rechaza la sesión. Si incluso la limpieza falla, queda un login sin privilegios, y el barrido de más abajo lo elimina.

Terminar una sesión

Una sesión termina de una de tres maneras: la persona se desconecta, pasa el límite de inactividad o pasa `ends_at`. Las dos últimas importan porque la gente deja pestañas abiertas. Una conexión abierta sobrevive a una credencial vencida (Capítulo 4), así que el gateway cierra él mismo las conexiones y después elimina el login. La misma función sirve para una persona que deja la empresa: un evento de baja del inicio de sesión único llama a `endSession`, y la duración máxima de la sesión es el límite para el caso en que ese evento nunca llegue.

```
function endSession(login, reason):
  // the single owner of a session's end; Chapter 8's expiry calls
  it too
  try:
    closeConnections(login)
  on CloseFailed:
    // a drop that fails on a live connection is retried by the
    sweep
    notify("platform", "connections not closed: " + login)
    // requestsOf returns only requests not yet marked revoked
    for each id in requestsOf(login):
      try:
        revokeRequestAccess(id, from: login)
      on LoginNotFound:
        // nothing is left to revoke
      on RevokeFailed:
        notify("platform", "session grants not revoked: " + login)
        return
    markRevoked(id)
    auditOrQueue({ action: "revoke", actor: "gateway",
                    subject: login, request: id })
  try:
    revokeIfExists("human_readonly", from: login)
    dropLoginIfExists(login)
  on DatabaseError:
    notify("platform", "session login not removed: " + login)
    return
  markSessionEnded(login, reason)
  auditOrQueue({ action: "session-end", subject: login, reason:
    reason })
```

Aquí `auditOrQueue` añade el registro al log de auditoría y, si el log es inalcanzable, lo pone en cola para reintentarlo y avisa al equipo de plataforma. La revocación y la eliminación ya ocurrieron para entonces, así que una caída del log retrasa la entrada y nunca la retirada del acceso. Un lo-

gin que ya no existe se trata como éxito: una sesión que la ruta de vencimiento del Capítulo 8 terminó antes se cierra aquí limpiamente, su registro se marca como terminado y no salta ninguna alerta.

El barrido llama a `endSession` para cada sesión que ya debería haber terminado.

```
function sweepSessions():
  // endSession reports a stuck login and returns, so one failure
  does not stop the loop
  for each s in sessionsPastEnd(before: now()):
    endSession(s.login, "expired")
  for each s in sessionsStuckOpening(before: now() - openingGrace):
    endSession(s.login, "abandoned")
  for each login in loginsWithoutRecord():
    endSession(login, "orphaned")
```

El orden es lo importante. Los grants de una solicitud se revocan a partir de su registro almacenado y se marcan como revocados solo después de que la revocación funcionó, y solo entonces se elimina el login, porque algunos motores se niegan a eliminar un login que aún tiene privilegios. Una revocación o una eliminación fallida avisa al equipo de plataforma y deja el registro para el siguiente barrido; una escritura de auditoría fallida se pone en cola y no detiene el bucle. El barrido da un período de gracia a una sesión que todavía está en “opening”, para no eliminar un login que solo tiene unos segundos. La comprobación de drift del Capítulo 3, que compara los logins que existen con los registros del gateway y los grants con las tripletas almacenadas, se ejecuta con su propia planificación, de modo que un job detenido no oculte al otro.

Qué base de datos, qué rol

Cada sesión aterriza donde le corresponde por su tipo. Una sesión rutinaria es un login `human_readonly` que se conecta a `ledgerly_replica_2`. Una sesión abierta para hacer una solicitud JIT o break-glass se abre en `ledgerly_primary`, y el Capítulo 8 muestra cómo se hace la solicitud desde esa sesión y cómo terminan con ella los grants. El gateway hace cumplir la regla de destino que el Capítulo 6 escribió como política de conexiones: es el gateway quien conoce el tipo, y el host de la base de datos quien rechaza un login pensado para otro.

Una persona puede tener dos sesiones abiertas a la vez, una rutinaria y otra con una solicitud. Mantendría eso posible y mantendría las sesiones visiblemente separadas, para que nadie use la privilegiada para una consulta casual. El costo humano es una segunda conexión en el cliente de la persona. El beneficio es que la sesión privilegiada es fácil de encontrar, corta y fácil de terminar.

Qué registra el gateway

Como el gateway de Ledgerly está en la ruta de datos, puede registrar cada sentencia que envía una sesión, junto con el inicio, el final y las filas afectadas. Almacenaría ese registro donde las personas a las que describe no puedan editarlo; el Capítulo 14 cubre dónde. Un registro tiene este aspecto:

```
statement record (illustrative):
  login:      sess-robin-0412
  person:     robin
  kind:       jit
  request:    req-0412
  fingerprint: UPDATE invoices SET status = ? WHERE id IN (?)  //
  literals redacted
  param_hash: 9c1e...
  rows:       12
  started_at: 03:21:07
  ended_at:   03:21:07
```

Los regímenes de datos de pago y de información financiera suelen preguntar quién tocó los datos de producción y por qué ruta; el registro de sesión es esa respuesta, y la interpretación corresponde a las personas de legal y cumplimiento de quien lee. Conviene conocer dos límites. Un log de sentencias dice lo que se pidió, no lo que se devolvió, y yo no almacenaría conjuntos de resultados completos, porque convertirían el log en una copia de los datos. El fingerprint lleva la forma de la sentencia y un hash de sus parámetros en lugar de sus valores, por la misma razón. Y un log que nadie lee registra sin proteger nada, así que decide quién lo mira y cuándo: una muestra semanal de las sesiones que tuvieron una solicitud JIT es un comienzo, y la revisión de accesos es la otra ocasión.

Capacidad y presupuesto de conexiones

Cada sesión humana es también una conexión a la base de datos, y el presupuesto del Capítulo 5 dio a las personas una porción fija, digamos veinte conexiones, a modo de ilustración. Las sesiones rutinarias de ingeniería aterrizan todas en `ledgerly_replica_2`, así que esa réplica es el host de la porción humana, y su límite de conexiones es donde por fin se hace cumplir el número. El gateway debería hacer cumplir él mismo esa porción: un límite por persona, un límite por tipo de sesión y una negativa clara cuando el presupuesto se agota. Una negativa en el gateway es un mensaje para una persona. Una negativa en la base de datos, cuando el presupuesto se ha agotado, puede desplazar a un servicio. Poner el límite en el gateway evita que el presupuesto humano se convierta en problema del servicio.

El laptop y la copia local

El gateway controla la ruta hasta la base de datos. No controla lo que pasa con los datos una vez que llegan a un laptop, y ahí vive una parte silenciosa del riesgo.

Dos cosas se quedan en el laptop. Una es la configuración de la conexión: el host guardado, el login guardado y, en el estado inicial, la contraseña guardada. La otra es lo que la persona hace con un resultado: un archivo exportado, una tabla pegada, una copia local de unos miles de filas para “jugar con ellas”. Cada una es una copia de datos en el sentido del Capítulo 6, un lugar donde viven datos de producción sin responsable y sin vencimiento.

Las trataría en tres capas, de la más barata a la más cara.

Quitar lo que el cliente puede recordar. Como las credenciales de sesión se generan para una sola sesión y nunca se muestran, un cliente de escritorio no tiene contraseña que guardar. La conexión guardada ahora contiene una dirección del gateway y el inicio de sesión único de la persona, y las entradas guardadas antiguas del primary se pueden borrar durante la migración, con una lista corta de qué borrar y dónde. Esta capa casi no cuesta nada y elimina la fuga más común.

Elegir valores por defecto que encajen con la tarea. El gateway puede abrir sesiones con valores por defecto de solo lectura y un statement timeout del Capítulo 5. También puede limitar las filas que devuelve una sentencia: cuenta las filas a medida que las transmite, cierra el cursor en el tope, se lo dice a la persona y escribe el evento en el registro de sentencias; una sesión que alcanza el tope repetidamente es lo primero que mira la muestra semanal.

Los valores por defecto que la persona puede levantar no siguen siendo valores por defecto. Un cliente puede enviar un `SET` que sube el timeout o desactiva el solo lectura, así que el gateway mantiene una lista de guarded settings y rechaza toda sentencia que los cambie. El costo es que algunos clientes envían tales sentencias al conectarse y se romperán; es un costo que hay que nombrar en la migración, y el Capítulo 9 cubre cómo decide el gateway qué sentencias rechazar. Los clientes que hacen auto-commit son menos peligrosos cuando la sesión no tiene privilegios de escritura con los que confirmar. Una consola de navegador, ejecutada por el propio gateway, da un paso más, porque no se descarga nada a menos que exista un botón de export y quede registrado. Esa consola es más lenta y menos agradable que un cliente nativo, y yo la ofrecería como opción para quienes solo miran y nunca necesitan sus propias herramientas, no como requisito.

Decidir explícitamente qué sale. Si un resultado puede exportarse desde una sesión es una regla aparte, y pertenece al trabajo de enmascaramiento y minimización del Capítulo 12 y a las rutas de export del Capítulo 13. El gateway puede aplicar la regla, pero no puede hacerla innecesaria. Una persona puede leer una pantalla, fotografiarla o reescribir un valor, y no conozco un control rentable para eso. El gateway reduce la cantidad de datos que salen por accidente y hace visibles los exports deliberados; no hace imposible la salida.

Nota: Si tus ingenieros necesitan extractos grandes para analizar, la respuesta suele no ser un límite de export mayor en el gateway. Es una copia de analítica enmascarada diseñada para ese propósito, que describen los Capítulos 6 y 12. El gateway es la herramienta equivocada para mover datos en masa.

Una puerta, un punto único de fallo

Un único punto de entrada es fácil de explicar, fácil de registrar y fácil de cerrar. También es un punto único de fallo, y conviene decirlo con claridad antes de recomendarlo. Si el gateway está caído, nadie puede llegar a la base de datos por la puerta principal, incluida la persona de on-call que más necesita la base de datos cuando ya se ha roto otra cosa.

No hay respuesta que elimine este costo. Hay respuestas que lo trasladan a un lugar que puedes manejar.

Mantén el gateway pequeño. Auténtica, crea logins, reenvía y registra. Cada funcionalidad extra es otra razón para que esté caído. Me resistiría a añadir transformación de resultados y una interfaz de aprobaciones al mismo servicio, y ejecutaría esas cosas como componentes separados a los que el gateway llama pero de los que no depende para iniciar una sesión de lectura. Donde una sentencia se comprueba antes de ejecutarse, esa comprobación también es un componente propio (Capítulo 9).

Ejecuta más de una instancia y pon el estado en un lugar duradero.

Dos instancias tras la misma dirección, compartiendo los registros de sesión, significan que un reinicio o un deploy defectuoso no termina todas las sesiones. El almacén de sesiones y el registro de auditoría también son dependencias, y su disponibilidad forma parte de la disponibilidad del gateway. Esa es la razón por la que la función de apertura de arriba rechaza una sesión cuando el log es inalcanzable: es una elección deliberada que ata el acceso humano al log, y debería ser una elección que el equipo hizo a propósito y dejó escrita.

Decide cuáles son los modos degradados. Hay dos fallos para los que planificar. Uno es que el gateway esté caído mientras las bases de datos están bien. El otro es que el registro de auditoría esté caído mientras el gateway funciona. Para cada uno, alguien responsable tiene que haber decidido de antemano si el acceso humano se detiene o continúa por una ruta más estrecha, y qué se registra mientras tanto. La decisión del libro para una caída del gateway es que no hay una segunda puerta: el acceso humano se detiene, break-glass incluido, ya que una sesión break-glass es una sesión del gateway con un rol más pesado. La manera de mantener eso poco frecuente es la redundancia de arriba, no un login de emergencia fuera del gateway. El Capítulo 8 da el razonamiento y lo que hace un equi-

po si decide otra cosa, y el Capítulo 15 describe cómo ensayar la caída. Para una caída del registro de auditoría, las lecturas en una réplica pueden almacenar entradas en un búfer y las escrituras no (consulta los trade-offs más abajo).

Trata el gateway como producción. Tiene un responsable, una ruta de on-call, un health check que una persona puede leer y un proceso de deploy con rollback. El equipo de plataforma es su responsable. Si el gateway es un proyecto paralelo de una sola persona ingeniera, el punto único de fallo es esa persona.

El gateway también concentra un riesgo distinto: un compromiso del gateway es un compromiso del acceso a todo lo que puede alcanzar. Por eso su login `ledgerly_access_admin` se custodia como se describió antes, y no es motivo para evitar el gateway: el estado inicial tiene una contraseña compartida, copiada a todo el equipo, en la misma posición.

El paso de Ledgerly de muchas rutas a una

Un equipo en la posición de Ledgerly no cambiaría todo en un solo día. Yo seguiría los pasos en un orden que mantenga disponible una ruta que funcione en cada momento.

1. **Inventariar las rutas.** Lista todas las maneras en que una persona puede llegar hoy a cualquier base de datos de Ledgerly: las rutas de la VPN, el bastion, las entradas guardadas en los clientes, cualquier script que se conecte con el login de administración. El recorrido de Alex al comienzo fue este paso, y debería terminar en una lista escrita que el equipo de plataforma mantenga.
2. **Levantar el gateway junto a las rutas antiguas.** Al principio crea logins de sesión en el primary, donde se definen los logins, les concede `human_readonly` y los conecta a `ledgerly_replica_2`, y nada más. Las rutas antiguas siguen funcionando. Unas pocas personas usan el gateway para su lectura diaria e informan de lo que resulta incómodo.
3. **Mover primero la lectura.** La lectura es la mayor parte de lo que hace la gente y la que menos riesgo supone probar. Una vez corregidas las partes incómodas, cierra la réplica a la VPN y borra las entradas guardadas que apuntaban a ella.

4. **Mover el primary al final.** El acceso directo al primary es en lo que la gente se apoya durante un incidente, así que lo cerraría solo cuando las rutas JIT y break-glass del Capítulo 8 funcionen y se hayan ensayado. Cerrarlo antes quitaría la ruta de emergencia antes de que el reemplazo esté listo.
5. **Eliminar el bastion antiguo y el alcance del login de administración.** Se apaga el bastion del contratista. El login de administración compartido deja de aceptarse desde redes humanas. Los servicios conservan sus propios logins hasta que el trabajo del Capítulo 4 los reemplace.
6. **Probar las denegaciones.** Para cada ruta anterior, alguien la intenta y confirma que falla. Una ruta denegada que nadie probó es una suposición.

El costo de la migración es sobre todo humano. Las personas tienen que aprender un nuevo flujo de conexión, y algunos de sus scripts se romperán porque suponían un login permanente. Planifica una ventana durante la cual ambos funcionen, y deja escrita la fecha en que se cierra la ruta antigua. Sin fecha, la ruta antigua sigue abierta, porque siempre parece más barato dejarla.

Trade-offs y costos

Un diseño más pequeño que puede bastar. Un gateway es un sistema nuevo en la ruta crítica, y si no puedes dotar de personal su ruta de on-call y sus deploys, un diseño más pequeño es el mejor comienzo. Si cuatro ingenieros llegan a una base de datos por una red privada, usando logins personales de vida corta emitidos por algo que tu equipo ya ejecuta, puede que ya tengas la mayor parte del valor. Dejaría escrito qué te haría cambiar de opinión: la llegada de un contratista, una segunda base de datos, una solicitud de grabaciones de sesión o el primer incidente en que el log no pudo decir quién lo hizo. El gateway es una respuesta a esas condiciones, no un hito.

Fallar cerrado tiene un precio. Rechazar una sesión cuando el registro de auditoría es inalcanzable mantiene completo el registro y convierte el almacén del log en una dependencia de cada sesión humana. Algunos equipos prefieren abrir sesiones de lectura con un búfer local y enviar las en-

tradas después. Es una elección defendible para sesiones de solo lectura en una réplica, y más débil para sesiones que escriben. Si la eliges, di a qué sesiones se aplica y monitorea el búfer.

El registro de sentencias es una copia de datos. El registro de fingerprints redacta los literales, pero aun así nombra tablas y columnas, y cualquier campo que quede sin redactar, como una nota de texto libre o un comentario aportado por el cliente, puede llevar un valor. Trata el log con la clasificación de lo más sensible que un registro podría contener, restringe quién lo lee y fija un período de retención del que alguien sea responsable.

Latencia y fidelidad. Un gateway en la ruta de datos añade un salto y puede no admitir todas las funciones de cliente ni todas las extensiones del protocolo. Algunas herramientas se comportan de forma extraña a través de un intermediario: las transferencias muy grandes, los cursores de larga duración y los flujos de autenticación poco habituales son los lugares de siempre. Prueba las herramientas que tu gente usa de verdad antes de prometer que la ruta antigua se cerrará.

Cuándo no usar un gateway. Un prototipo de un solo servicio sin datos de clientes no necesita uno. Un sistema cuyo único acceso humano es una consola gestionada que ya ofrece identidad, credenciales de vida corta y registro puede no necesitar una segunda capa delante. Un bastion es una respuesta razonable cuando la necesidad real es un único lugar donde cerrar la puerta de red y cuando el sistema protegido tiene su propia autenticación fuerte por persona. En ambos casos se aplican las mismas preguntas: quién es la persona, cuánto dura el acceso y dónde está el registro.

Errores comunes

- **Tratar la VPN como la capa de identidad.** Es tentador porque la VPN ya sabe quién está en ella y todo el mundo la tiene. Demuestra pertenencia a la red, no qué login de base de datos se usó ni qué hizo. Mantén la VPN como la manera de llegar al gateway y añade una capa que asocie una persona a cada sesión.
- **Poner un bastion delante de la base de datos y dar el trabajo por hecho.** Ordena la red y deja el login compartido y las sentencias invisibles. Elige un bastion por lo que hace, una única puerta de red que se puede cerrar, y busca la identidad y el registro en otra parte.

- **Dejar abierta la ruta antigua durante la migración.** Parece prudente mantener ambas rutas un tiempo, y lo es, pero sin una fecha de cierre la antigua sobrevive a la nueva. Escribe la fecha, anúnciala y comprueba que la ruta antigua falla después.
- **Abrir una sesión cuyo final nadie hace cumplir.** Un límite de sesión que solo existe en la documentación es un grant permanente con un nombre más largo. Termina las sesiones en `ends_at`, cierra las conexiones además de eliminar el login y ejecuta un barrido que encuentre las sesiones y los logins que no deberían existir.
- **Saltarse la decisión sobre el modo degradado.** Un equipo que construye un gateway sin decidir qué pasa cuando está caído decidirá bajo presión, normalmente reabriendo la vieja ruta compartida “solo un minuto”. Decide de antemano la ruta de emergencia, déjala escrita y ensáyala.
- **Registrarlo todo y no leer nada.** Capturar cada sentencia parece diligencia. Sin un responsable que la muestree y una revisión de accesos que la examine, el log es una copia de datos sin propósito. Decide quién lo lee y qué busca.
- **Leer la lista de red como más precisa de lo que es.** En una red de clúster compartida la regla no distingue un servicio de otro, y quien revisa y supone que sí lo hace aprobará una ruta más ancha de lo que parece. Escribe en su lugar qué control separa a quienes llaman.

Ideas clave

- El alcance de red y el acceso a la base de datos son preguntas separadas, y un diseño que responde a ambas con el mismo control no puede decir qué persona hizo qué.
- Un bastion vuelve singular la ruta de red pero no entiende el protocolo de la base de datos, así que por sí solo deja el login compartido y las sentencias invisibles.
- Un access gateway autentica a una persona mediante el inicio de sesión único, crea un login de sesión que lleva esa identidad y lo elimina cuando la sesión termina, que es lo que hace que el log de la base de datos nombre a una sola persona.

- La regla de red que permite que solo el gateway llegue a los hosts de la base de datos es el control que hace del gateway la única puerta, y debería ser una lista corta que una revisión de accesos pueda leer.
- Un único punto de entrada es un punto único de fallo, así que el gateway debería ser pequeño, ejecutarse en más de una instancia, tener al equipo de plataforma como responsable y ir acompañado de una decisión sobre el modo degradado tomada antes del incidente.
- El gateway no puede controlar lo que una persona hace con los datos que han llegado a un laptop; puede quitar credenciales guardadas, fijar valores por defecto de solo lectura y hacer visibles los exports, y el resto pertenece al enmascaramiento y la minimización.
- Migra primero la lectura, cierra el primary al final, fija una fecha de cierre para cada ruta antigua y comprueba que las rutas denegadas están denegadas.

El gateway da a cada persona una sesión con un nombre y una lectura por defecto, pero leer rara vez es todo el trabajo. El próximo capítulo examina qué ocurre cuando alguien necesita más que `human_readonly` durante un tiempo limitado: el acceso just-in-time, el acceso break-glass y lo que la aprobación cuesta a quienes esperan por ella.

Capítulo 8 — Acceso just-in-time y break-glass

Imagina una noche de martes en Ledgerly, poco después de las tres de la madrugada. Una alerta avisa de que un lote de facturas lleva atascado en el estado `sending` desde que se reinició `invoice-worker`, y los correos de recordatorio de la ejecución de la mañana no saldrán hasta que se libere el lote. Nada está expuesto y nada se ha perdido; el costo del problema es un retraso. Robin, el on-call de backend, abre una sesión rutinaria a través del access gateway. La sesión es de solo lectura y corre en una réplica, que es lo que Robin quiere durante los primeros diez minutos: mirar, contar, entender.

La mirada confirma el problema, y la corrección necesita un `UPDATE` sobre una columna de `invoices`. Robin deja por escrito la corrección de una sola sentencia y presenta una solicitud para ella. Quien debe aprobarla es el on-call secundario, que está dormido, y nadie ha dado ack al page. Robin tiene un laptop, un runbook y una decisión que tomar: esperar una respuesta o buscar la manera de saltarse la pregunta. En el Ledgerly de antes la decisión no habría existido, porque la contraseña de administración compartida estaba a un pegado de distancia. En el Ledgerly nuevo hay que diseñar el atajo, y también el camino de vuelta.

Este capítulo añade los siguientes pasos del estado objetivo del ejemplo que seguimos: las personas no tienen acceso de escritura permanente, las escrituras llegan como grants just-in-time dentro del techo del Capítulo 3, y existe un rol de emergencia que se usa pocas veces y siempre se revisa. En las cinco partes del libro, es sobre todo el Grant: quién aprueba qué, para qué sentencia, por cuánto tiempo. Responde a la pregunta de Robin: ¿cómo aparece el acceso cuando hace falta, cómo termina con la tarea y qué ocurre cuando el camino normal es demasiado lento?

El problema y por qué es difícil

El acceso permanente es la promesa de que nadie tendrá que volver a pedir nada. Es cómodo durante un mes, y después nadie recuerda qué cubría la promesa. Las personas cambian de equipo y conservan sus grants; un rol ampliado para un incidente se queda ancho. Cuando llega una revisión de accesos (Capítulo 14), quien revisa se encuentra con grants sin ninguna tarea asociada, y la respuesta a “¿sigue haciendo falta?” es “probablemente”. El costo de un grant permanente son todos los días en que no se usa, porque sigue cargando con el radio de impacto de todo lo que cubre.

El reemplazo evidente es el **acceso just-in-time (JIT)**: acceso concedido para una tarea concreta y por un tiempo limitado. Es fácil de enunciar y fácil de hacer mal, con sutileza, en cinco puntos.

Alcance. La implementación fácil concede un rol a un login durante una hora. Si ese rol es `human_jit_write`, el login recibe el techo completo: derechos de actualización sobre seis tablas para una tarea que necesitaba una columna de una sola, y el grant es un grant permanente con un temporizador. El alcance tiene que vivir en los propios grants, solicitud por solicitud.

Vencimiento. En los motores comunes los grants no vencen por sí solos. Algo tiene que retirarlos, normalmente un job programado, y un job puede detenerse, perder su credencial, fallar en un login y seguir con los demás, o ejecutarse contra la lista equivocada.

Aprobación. Que una segunda persona lea cada solicitud tiene un costo humano: a alguien se le despierta, alguien espera. Si las aprobaciones son demasiado lentas, la gente encuentra el camino más rápido, que es la contraseña compartida que el libro intenta retirar. Si quien aprueba responde siempre en segundos, la aprobación deja de reducir la exposición.

La emergencia. Un esquema de acceso sin una forma autorizada de moverse con rapidez acaba generando una no autorizada. La cuestión es si el camino rápido se diseñó, se registró y se revisó, o se improvisó.

La operación. Un grant nombra una tabla y una columna y no dice nada de qué filas ni de qué sentencia. Quien aprueba al leer “UPDATE sobre `invoices`, `status`” ha aprobado una capacidad, y todo lo que se ejecute

dentro de ella queda fuera de la aprobación. Para que aprobar signifique “una segunda persona vio lo que va a pasar”, la solicitud tiene que nombrar la operación y el grant tiene que derivarse de ella.

Una política que dice “el acceso debe ser temporal” no le dice al sistema qué hacer cuando el job de revocación está caído. El resto del capítulo conecta grants, registros y temporizadores para que lo haga.

Acceso just-in-time: solicitudes que nombran una operación

Una solicitud JIT se compone de cinco elementos, cada uno presente por uno de los fallos anteriores. Cada uno lleva la etiqueta de la parte del modelo del libro a la que sirve.

- El techo (Grant), `human_jit_write`, define lo máximo que una solicitud puede pedir. Nunca se concede a un login (Capítulo 3).
- Una solicitud de escritura (Grant) nombra la corrección revisada para la que sirve, en `fix_id`. Sus tripletas (privilegio, tabla, columnas) se derivan de esa corrección, dentro del techo, y se guardan en `access_request_privilege`.
- La aprobación (Grant) es de la sentencia, y el grant hace cumplir sus límites: quien aprueba lee la sentencia, el gateway concede exactamente las tripletas derivadas al login de sesión de quien solicita, y el conjunto de sentencias aprobado se registra como `approved_hash`. El vencimiento revoca exactamente esas tripletas, leídas del mismo registro.
- Las escrituras de una sesión JIT (Ejecución) se reenvían únicamente desde el runner, que rechaza todo lo que no coincide con `approved_hash`.
- Una conciliación independiente (Registro) compara los grants que tiene la base de datos con las solicitudes que los justifican, de modo que algo distinto del propio job nota que el job se detuvo.

Los tomaré en el orden en que los encuentra una solicitud.

Qué contiene una solicitud

Una solicitud se hace desde la sesión que la va a usar, y esa sesión corre en el primary. Una sesión lleva como máximo una solicitud viva; una tarea que sobrepasa su ventana continúa en una sesión nueva con una solicitud nueva. La investigación rutinaria se queda en la réplica con `human_readonly` (Capítulo 6), y el Capítulo 7 dice quién puede abrir una sesión dirigida al primary. El gateway sabe qué login corresponde a cada persona, así que nadie escribe un principal ni un login.

Esta es la forma de la solicitud de Robin, antes de que nadie la haya respondido:

```
access_request:
  id:          "req-0412"
  kind:        "jit"
  requester:   "robin"
  requester_login: "sess-robin-0412"
  approver:    null
  fix_id:      "fix-0093"
  approved_hash: null           // set at approval, never by the
runner
  reason:      "alert-7731: invoices stuck in sending"
  expires_at:  now() + 1 hour   // illustrative window
  status:      "pending"

access_request_privilege:           // derived from fix-0093, not
typed
  ("UPDATE", "invoices", "status")
```

Hay tres cosas que conviene notar. El motivo apunta a algo comprobable, una alerta o un ticket. El grant es estrecho, y Robin no necesita ninguna tripleta `SELECT` porque la sesión ya tiene `human_readonly`. Y Robin no escribió la tripleta: se derivó de `fix-0093`, de modo que el grant y la sentencia que lee quien aprueba no pueden discrepar. Una solicitud que contiene solo tripletas `SELECT`, por ejemplo para leer sin enmascarar una columna Confidential, no tiene corrección ni hash.

Un grant restringe privilegios y columnas, no filas. El `UPDATE` de arriba puede cambiar `status` en todas las facturas que la sesión puede ver. Un grant JIT acota por columna y por tiempo hasta dónde puede llegar un error; lo que lo acota por fila es la sentencia que leyó quien aprueba y que ejecuta el runner, con la disciplina de contar primero del Capítulo 10.

Vincular la solicitud a la sentencia

La corrección es un conjunto de sentencias revisado, con su filtro y su conteo esperado de filas, en la forma que describe el Capítulo 10, y la solicitud la nombra por `fix_id` y lleva su `approved_hash`. El hash proviene del registro de merge de la corrección o del registro de la solicitud aprobada, y nunca de la entrada del runner; en una sesión de escritura JIT el gateway reenvía las escrituras únicamente desde el runner. Una solicitud que contiene solo tripletas `SELECT` no tiene corrección, hash ni runner, y quien aprueba lee la lista de columnas. El Capítulo 9 enuncia la regla completa y explica cómo comprueba el gateway cada sentencia.

Comprobar una solicitud

El gateway comprueba la forma de la solicitud antes de pedirle a nadie que la apruebe, y vuelve a comprobarla, junto con quien aprueba, en el momento de conceder el grant.

```
function checkRequestShape(request):
    // runs before anyone is asked to approve
    // live = approved and not yet revoked; this request is not yet
    counted
    if request.status != "pending" or
    hasLiveRequest(request.requester_login):
        raise GrantRefused
    if request.expires_at is null or request.expires_at <= now():
        raise GrantRefused
    // a window longer than the gateway allows is refused, not clamped
    if request.expires_at > now() + maxJitWindow:
        raise GrantRefused
    for each p in request.privileges:
        // an empty column list is never read as "all columns"; table-
        level
        // triples exist only on engines using the secrets-tables
        fallback
        if p.privilege != "DELETE" and isEmpty(p.columns):
            if not tableLevelAllowed(p.table):
                raise GrantRefused
    // refuses a table outside the ceiling and any Restricted column
    if not withinCeiling("human_jit_write", request.privileges):
        raise GrantRefused
    checkOperationBinding(request)
```



```

function checkOperationBinding(request):
    if hasWriteTriple(request.privileges):
        fix = fixRecord(request.fix_id)    // null when fix_id is null
        if fix is null:
            raise GrantRefused
        // the triples come from the fix; a typed list that differs is
        refused
        if request.privileges != triplesFromFix(fix):
            raise GrantRefused
        else if request.fix_id is not null or request.approved_hash is not
        null:
            // a SELECT-only request has no fix and no hash
            raise GrantRefused

function checkJitRequest(request):
    // runs again at grant time, once an approver is named
    checkRequestShape(request)
    if request.approver is null or request.approver ==
    request.requester:
        raise SelfApprovalNotAllowed
    if request.fix_id is not null:
        if request.approver == fixRecord(request.fix_id).author:
            raise SelfApprovalNotAllowed

```

Las líneas de autoaprobación son donde el four-eyes se hace cumplir en el código: quien solicita, o quien escribió la corrección, no puede ser quien responda. Una solicitud pendiente no tiene aprobador, y por eso la comprobación de forma omite esas líneas; el aprobador se rellena en el momento de la aprobación, y la comprobación completa corre al conceder el grant.

Rechazaría una ventana demasiado larga en lugar de acortarla en silencio: una ventana acortada hace que quien solicita crea que tiene una hora cuando tiene veinte minutos, y la sorpresa llega a mitad de la reparación.

`withinCeiling` es donde el techo del Capítulo 3 se vuelve exigible. Rechaza una tripleta cuya tabla está fuera del techo, cuyo privilegio no se ofrece en esa tabla, o cuya lista de columnas incluye `iban`, `account_holder_name` o `token_value`. Rechaza `INSERT` y `DELETE` sobre `bank_accounts` y `payment_tokens`, porque una fila nueva o eliminada establece o destruye valores Restricted. Rechaza toda tripleta, salvo un `DELETE`, que no tenga lista de columnas, excepto una tripleta a nivel de tabla en un motor que use el fallback de tablas de secrets. Y rechaza cualquier cosa que toque `ledger_entries`, `audit_events` o los registros de solicitudes de acceso, que ningún rol humano escribe.

Una cosa se queda dentro del techo a propósito. `UPDATE (customer_id)` sobre `bank_accounts` o `payment_tokens` pasa `withinCeiling`, porque `customer_id` solo vincula una fila con su propietario y no es `Restricted`. Aun así, mover la fila a otro cliente cambia de quién es esa cuenta o ese token, así que la política de ejecución lo envía a la ruta de producto (Capítulo 9). La lista de verificación de quien aprueba lo detecta a mano.

Quién aprueba, por cuánto tiempo y con qué alcance

Una solicitud JIT tiene tres variables: quien aprueba, la ventana y el alcance.

Quien aprueba es una persona que entiende las tablas de la solicitud y que no es quien solicita ni quien escribió la corrección. En Ledgerly, la persona natural para aprobar sobre `invoices` es otra de backend, normalmente el on-call secundario. Para cualquier cosa sobre `payments` o `payment_tokens`, la persona natural es alguien del equipo responsable de `payments-service`, porque ese equipo sabe qué desencadenará aguas abajo un `status` modificado allí. El equipo de plataforma opera las herramientas de acceso, y yo no lo haría aprobador por defecto de datos que no son suyos: se convierte en el cuello de botella, y en un aprobador que no puede juzgar el cambio.

```
function chooseApproverPool(request):
    owners = emptySet()
    for each p in request.privileges:
        owners.add(owningTeam(p.table))
    pool = eligibleApprovers(owners)
    // neither the requester nor the author of the fix may answer
    pool.remove(request.requester)
    if request.fix_id is not null:
        pool.remove(fixRecord(request.fix_id).author)
    if pool is empty:
        raise NoApproverAvailable
    return pool
```

Un pool vacío falla de forma visible y nunca recurre a uno más amplio. Por lo demás, el orden del pool es el orden de escalado: el page va a la primera persona, luego a la siguiente, dentro de la misma solicitud pendiente, que solo se rechaza cuando llega a `expires_at` sin respuesta.

La ventana debe durar lo que dura la tarea típica, no lo que dura la peor. Supongamos que una corrección de datos suele llevar veinte minutos, que el formulario ofrece una hora y que el máximo del gateway es de cuatro horas. Ampliar un grant es una solicitud nueva con una aprobación nueva, no una edición de la anterior, porque una edición permitiría estirar una única aprobación indefinidamente. Cuando una reparación sobrepasa su ventana, la sesión termina y cualquier transacción abierta hace rollback con ella, de modo que una reparación escrita como una sola transacción (Capítulo 10) pierde su trabajo limpiamente en lugar de dejar la mitad aplicada.

El alcance son las tripletas y la sentencia de la que proceden. Quien aprueba lee ambas, y la pantalla de aprobación muestra las tripletas como los grants en que se convertirán: “UPDATE sobre `invoices (status)`, hasta las 04:12”. Una solicitud que nombra una columna Confidential en un `SELECT` muestra esa columna sin enmascarar, y la pantalla lo dice con palabras, no solo en la lista de columnas.

La aprobación ocurre en la propia pantalla del gateway, mediante inicio de sesión único con una verificación multifactor reciente en el momento del clic, y no como reacción bajo un mensaje de chat: la aprobación es el control, así que lo que la registre debe saber quién la dio.

Qué comprueba quien aprueba

La decisión de quien aprueba es más acotada que “¿es buena idea?”: aprobar la sentencia y dejar que el grant haga cumplir sus límites. Esta lista de verificación refleja, desde el lado del acceso, la lista de quien revisa la corrección en el Capítulo 10.

1. ¿El motivo es una alerta o un ticket comprobable, y sigue abierto?
2. ¿La sentencia que aparece en la pantalla hace lo que dice el motivo?
Lee la sentencia, no el resumen que hace de ella quien solicita.
3. ¿El filtro acota las filas, y el conteo esperado coincide con lo que vio quien solicita?
4. ¿Las tripletas derivadas cubren la sentencia y nada más? Con el fallback de tablas de secrets, “nada más” significa “todas las columnas no Restricted de la tabla”, y la pantalla lo dice.

5. ¿La ventana dura aproximadamente lo que esta tarea y está por debajo del máximo?
6. ¿La sentencia toca `customer_id` en `bank_accounts` o `payment_tokens`, o cualquier columna Restricted? Entonces no es una corrección de datos JIT. Envíala a la ruta de producto y rechaza la solicitud.
7. ¿No soy ni quien solicita ni el autor, y me resultaría fácil explicar esta aprobación a quien revise la semana que viene?

Quien aprueba y no puede responder a una de estas preguntas se las plantea a quien solicita mientras la solicitud sigue pendiente. El silencio no es aprobación, y una lista de columnas tampoco.

Conceder el grant

La aprobación es el momento en que cuatro registros tienen que coincidir: el registro de solicitud del gateway, la entrada del registro de auditoría, los grants de la base de datos y la sesión. Ninguna transacción los abarca, así que el orden importa, y también lo que deja cada fallo.

```

function grantJitAccess(request):
    checkJitRequest(request)

    // the hash comes from the records, as "Binding the request"
    explains
    hash = approvedHashFor(request)
    // the approved record comes first: the revoke job reads it, so a
    grant
    // can never exist without its triples and its end
    markApproved(request.id, approved_hash: hash)

    // audit next: if the log is unreachable, no grant is made
    try:
        auditLogAppend({ actor: request.approver, action: "grant",
                        subject: request.requester_login,
                        request: request.id, fix: request.fix_id,
                        approved_hash: hash, triples:
request.privileges,
                        expires_at: request.expires_at })
    on AuditLogUnavailable:
        markRefused(request.id)
        raise GrantRefused

    // the session's end follows the approved window, as "Ending the
    grant" says
    setSessionEnd(request.requester_login, request.expires_at)

    // separate systems, no shared transaction: a failure part-way
    leaves an
    // approved record with no grant or only some triples
    for each p in request.privileges:
        grant(p.privilege, on: p.table, columns: p.columns,
            to: request.requester_login)

    // the window may end between the check and the grants: end it now
    if request.expires_at <= now():
        revokeRequestAccess(request.id, from: request.requester_login)

```

Léelo por sus modos de fallo. Si el registro de auditoría está caído, la solicitud se rechaza y no se concede nada: un grant que no se puede registrar no se puede revisar, y prefiero detener una reparación a crear una invisible. Si el gateway muere después del registro y de la entrada del log pero antes de los grants, la solicitud queda aprobada y sin efecto, lo que no expone nada. Si muere después de dos de tres tripletas, el login tiene dos, el registro tiene tres, y la revocación posterior retira las tres, ya que revocar una triplete que nunca se concedió es inofensivo.

Las últimas líneas tratan una condición de carrera. La ventana no se reinicia con la aprobación, así que la comprobación de forma rechaza una aprobación que llega después de `expires_at`; lo que queda es una aprobación que pasa un instante antes de que la ventana se cierre, y esas líneas retiran sus grants de inmediato.

La entrada de auditoría lleva la corrección y el hash junto con las tripletas, de modo que quien aprueba, la sentencia y la ventana quedan en un solo registro. Los regímenes de datos de pago y de informes financieros suelen preguntar quién aprobó un cambio en datos de producción, con qué fundamento y por cuánto tiempo, y este registro responde a las tres cosas; la interpretación corresponde a los equipos legal y de cumplimiento.

Terminar el grant

Un grant termina en tres capas, y de ninguna se confía como la única.

La primera capa es la sesión. El gateway la termina en `expires_at` y elimina el login, sin ningún job programado de por medio: el reloj que acota el acceso pertenece al componente que lo creó. Una sesión dirigida al primary se abre con el tope de dos horas del Capítulo 7, así que, al conceder un grant, desde `grantJitAccess` o `declareBreakGlass`, el gateway restablece el `ends_at` de la sesión al `expires_at` de la solicitud, y `endSession` del Capítulo 7 cierra la sesión en ese momento; por tanto, una ventana de más de dos horas no se corta.

```
function endSessionAtExpiry(session):  
    // one live request per session, so its end is the session's end  
    waitUntil(session.request.expires_at)  
    // never raises; a failure leaves the record for the next sweep  
    endSession(session.login, "expired")
```

El cierre en sí es `endSession`, el único lugar que cierra conexiones, revoca lo que concedieron las solicitudes de la sesión y elimina el login, y `sweepSessions` llama a la misma función (Capítulo 7).

La segunda capa es el job de revocación, que lee el mismo registro y retira lo que concedió la solicitud. `revokeRequestAccess` deriva su trabajo de `access_request_privilege`: para una solicitud JIT, revoca cada tripleta

almacenada; para una solicitud break-glass, revoca el rol `human_breakglass`. No lee una lista aparte que pudiera discrepar, y trata un login inexistente como ya revocado.

```
function revokeExpiredGrants():
  // expiredGrants: kind jit or breakglass, status approved,
  expires_at
  // passed, not yet revoked
  for each item in expiredGrants(before: now()):
    try:
      // revokes what the request granted: its stored triples
    for "jit",
      // the role human_breakglass for "breakglass"
      revokeRequestAccess(item.request_id, from: item.login)
    on LoginNotFound:
      // the login went with the session; nothing is left to
    revoke
      on RevokeFailed:
        // an expired grant still live is an incident; retried
    next run
      notify("platform", "expired grant still active: " +
    item.id)
      continue
      markRevoked(item.request_id)
      try:
        auditLogAppend({ action: "revoke", actor: "revoke-job",
          subject: item.login, request:
    item.request_id })
      on AuditLogUnavailable:
        // the revoke is done; the log catches up from the retry
    queue
      queueForRetry(item.request_id)
      notify("platform", "revoke not yet logged: " + item.id)
```

La revocación continúa cuando el registro de auditoría está caído. La asimetría es deliberada: un grant sin entrada en el log es una exposición, mientras que una revocación sin ella es un hueco que se puede rellenar más tarde.

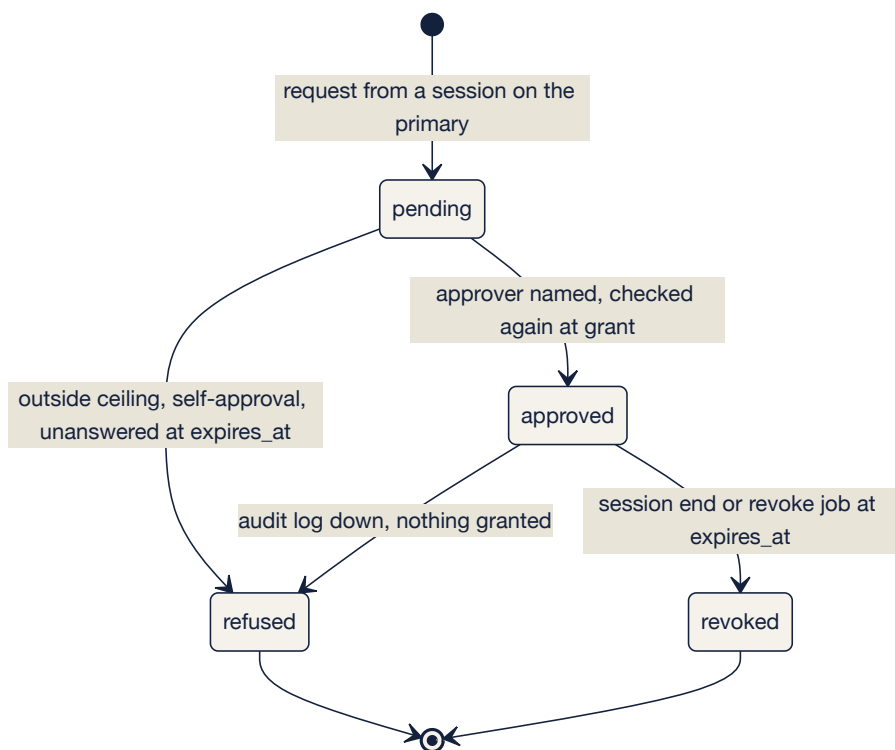
La tercera capa es la conciliación, y es una cosa de otra naturaleza. No retira acceso. Pregunta, desde fuera, si las dos primeras capas hicieron su trabajo:

```
function checkOverdueGrants():
  // scheduled separately from revokeExpiredGrants, so one failure
  // cannot hide the other
  if count(expiredGrants(before: now() - graceMinutes)) > 0:
    notify("platform", "grants past expiry and not revoked")
  if now() - lastSuccessfulRun("revokeExpiredGrants") > maxRunAge:
    notify("platform", "revoke job has not completed recently")
```

Junto a ella, la comprobación de drift del Capítulo 3 lee los grants que la base de datos tiene realmente y los compara con las tripletas no vencidas de las solicitudes. Un grant directo sin ninguna solicitud viva detrás, un login que es miembro del techo, un miembro de `human_breakglass` sin solicitud break-glass abierta: cada uno se señala, incluso cuando el job de revocación informa de éxito. La verificación de atrasos vigila los registros sobre los que el job debería haber actuado; la comprobación de drift vigila la base de datos que el job debería haber cambiado. Un job que informa “hecho” sin cambiar nada solo lo detecta la segunda.

La conciliación no repara nada por sí sola, y yo lo mantendría así: una revocación automática de grants desconocidos es una forma de romper producción de noche. Notifica al equipo de plataforma, y una persona decide.

El estado por el que pasa una solicitud es pequeño, y dibujarlo muestra dónde actúa cada capa. Fíjate en las salidas de `approved`: una revocación, o un rechazo cuando el registro de auditoría está caído y no se concedió nada; de `revoked` no se vuelve a ninguna parte.



Autoaprobación, y sin aprobación automática para el on-call

La autoaprobación se rechaza siempre. `checkJitRequest` lanza `SelfApprovalNotAllowed` y nada lo anula, ni la antigüedad ni la urgencia. El caso urgente tiene otro camino, que es la segunda mitad de este capítulo.

La aprobación automática es el otro atajo tentador, porque la fricción es máxima de noche. Para las lecturas yo no la construiría: todos los logins de sesión ya tienen `human_readonly`, y lo único que añade un `SELECT JIT` es una columna `Confidential` mostrada sin enmascarar, que es justo lo que debe mirar quien aprueba. Una escritura cambia aquello sobre lo que actuarán los servicios a continuación, y una máquina no puede saber si `status` es la columna correcta. Mantén las escrituras con un aprobador humano y dedica el esfuerzo a que ese aprobador sea fácil de alcanzar.

Si más adelante añades una aprobación automática, para una lectura que de verdad aporte algo nuevo, la regla debe exigir una lista de columnas explícita, y una lista vacía o ausente debe rechazar la solicitud de forma visible. Con el fallback de tablas de secrets, una lectura a nivel de tabla no

tiene ninguna lista de columnas, así que una regla que trate “ninguna columna Confidential nombrada” como aceptable aprobaría todas las columnas Confidential de la tabla.

Solicitudes desde la automatización, y la política como código

Un job, un script de runbook o un agente de IA que quiere escribir es un principal como cualquier otro: abre su sesión bajo su propia workload identity, nombra una corrección y espera a un aprobador humano, y el registro nombra tanto a la automatización como a la persona que la activó (el Capítulo 9 explica qué puede hacer dentro de una sesión). La lógica de aprobación es en sí misma código. `withinCeiling`, `chooseApproverPool`, `maxJitWindow` y las comprobaciones de forma viven en el repositorio, pasan por revisión y forman parte del baseline de la comprobación de drift; un techo editado a mano en una consola es el problema original un nivel más arriba.

Casos de abuso

Hay tres casos que los controles anteriores acotan y ninguno elimina. La colusión entre quien solicita y quien aprueba queda reducida a las columnas, la sentencia y la ventana del registro, y la pareja es visible en la entrada de auditoría; la revisión de accesos (Capítulo 14) es donde aparece una pareja que se aprueba mutuamente mucho más a menudo de lo que daría el azar. Un aprobador víctima de ingeniería social (“aprueba req-0412 ya, el lote está bloqueando la nómina”) se enfrenta con la lista de verificación: lee la sentencia en la pantalla del gateway, comprueba la alerta del motivo y no responde a nada que llegue por otro canal. Un token de inicio de sesión único secuestrado no puede escribir en una sesión JIT, porque el gateway reenvía las escrituras únicamente desde el runner y el runner rechaza un hash que nunca se aprobó. El riesgo residual está en las lecturas dentro de las tripletas concedidas y en el break-glass, que no tiene aprobador; lo que lo acota son la verificación reforzada en la declaración y la alerta.

Acceso break-glass

Break-glass es un acceso de emergencia con un conjunto de logging definido y una revisión obligatoria. El nombre viene del interruptor de incendios tras un cristal: al alcance de cualquiera que lo necesite, y romperlo es

visible para todos.

En el modelo del libro es el rol `human_breakglass`, y es el único rol humano que lee columnas `Restricted`. Lee todas las tablas y escribe `INSERT` y `UPDATE` en todas las tablas excepto las dos `append-only`; en `bank_accounts` y `payment_tokens` escribe únicamente `UPDATE`, y solo en columnas no `Restricted`. Nunca escribe en `ledger_entries` ni en `audit_events`, porque ningún rol humano lo hace. No es una ruta para correcciones rutinarias, y tampoco es una ruta de corrección de datos: una corrección de datos se planifica, se revisa y se acota (Capítulo 10), y el `break-glass` no es nada de eso por diseño.

Una solicitud `break-glass` es autodeclarada. No tiene aprobador. La persona en la sesión dice que existe una emergencia, da un motivo, y el `gateway` marca la solicitud como aprobada y concede el rol al mismo login de sesión hasta `expires_at`. El mismo job de revocación y el mismo temporizador de sesión lo retiran.

El aprobador es todo el control en el JIT, así que un rol de emergencia sin él puede parecer extraño. El `break-glass` existe para los casos en que el pool de aprobadores no ha respondido, y un paso de aprobación reproduciría ese fallo. El control pasa de estar antes del acceso a estar alrededor de él: una alerta en la declaración, el conjunto de logging que sigue, una ventana corta y fija, y una revisión que viene después sin excepción.

```

function declareBreakGlass(session, reason):
    // a hijacked token alone must not be enough to declare
    requireFreshMultiFactor(session.principal)
    // one live request per session: a break-glass declaration
    supersedes it
    supersedeLiveRequest(session.login)
    // the window is the gateway's fixed maximum; there is no argument
    for it
    window = breakGlassMaxWindow
    request = newRequest(kind: "breakglass", requester:
session.principal,
                                requester_login: session.login, approver:
null,
                                reason: reason, expires_at: now() + window)
    markApproved(request.id)
    try:
        auditLogAppend({ actor: session.principal, action:
"breakglass",
                                subject: session.login, request: request.id,
                                expires_at: request.expires_at })
    on AuditLogUnavailable:
        markRefused(request.id)
        raise GrantRefused
    setSessionEnd(session.login, request.expires_at)
    grant(human_breakglass, to: session.login)
    // paged after the grant: a paging outage must not block an
    emergency;
    // the review queue below is the net for a page that never arrived
    queueReview(request.id, reviewer: "not-the-requester")
    notify("security", "break-glass declared: " + request.id)
    notify("platform", "break-glass declared: " + request.id)

```

Tres decisiones de ese bloque merecen un comentario.

La ventana la fija el gateway, en dos horas en la ilustración de Ledgerly. No hay una ventana más larga disponible a solicitud, porque “necesito un poco más” convierte una excepción en un hábito. Si la emergencia sobrepasa la ventana, la persona vuelve a declarar y la segunda declaración vuelve a alertar; esa es la fricción buscada.

La comprobación del registro de auditoría es la misma que en el JIT. Si el log está caído, la declaración se rechaza. Acepto ese costo, porque una sesión break-glass sin registro es lo único que una revisión no puede reconstruir, y la revisión es lo que hace tolerable el rol. Qué hace el on-call cuando el camino normal y el log están caídos a la vez pertenece al Capítulo 15.

La alerta es una notificación a los contactos de seguridad y al equipo de plataforma de que alguien ha abierto el acceso más amplio que tiene Ledgerly, no una solicitud de permiso. En un sistema sano es lo bastante rara como para que se lea; si salta cada semana dejará de leerse, y el break-glass se está usando para algo que merece un camino diseñado.

El conjunto de logging del break-glass

“Logging intensivo” no es un estándar hasta que se define, así que para una sesión break-glass el conjunto de logging son estas cuatro cosas:

- Cada sentencia, como un fingerprint con sus literales con redaction y un hash de los parámetros, con el inicio, el fin y las filas afectadas. La redaction importa más aquí: esta sesión lee columnas Restricted, y un literal en el log las copiaría a un lugar sin protección a nivel de columna.
- Las entradas a medida que ocurren. Cada sentencia se añade al registro de auditoría mientras se ejecuta, no se recoge al final, de modo que una sesión que se corta deja igualmente su registro.
- Una nota de una línea en cada escritura. El gateway rechaza una escritura sin ella, lo que cuesta segundos y le da a quien revisa la intención junto a la sentencia.
- El mismo registro que cualquier otra sesión: el login, la persona, la solicitud y el motivo, de modo que los tres registros del Capítulo 14 se puedan unir.

Queda un riesgo, y difiere según el estado. Hoy, sin gateway o con uno que solo intermedia la credencial, una sesión break-glass es el único lugar donde sigue siendo posible un `UPDATE` sin acotar sobre una tabla en la que puede escribir. La ventana, los conteos de filas, las entradas y la revisión acotan el daño y lo dejan al descubierto; no lo previenen. En el estado objetivo, el gateway del Capítulo 9 ejecuta las sesiones de escritura con auto-commit desactivado y un row cap, su matriz deniega una escritura sin acotar en este lane, y se rechaza una escritura sin nota. Lo que sigue siendo posible es una escritura que cabe dentro del tope y aun así afecta a las filas equivocadas. El tope es aquí lo más difícil de dimensionar, porque una emergencia no es por definición la tarea planificada; hasta que la política fije uno, la escritura sin acotar es un riesgo aceptado, y la razón de que exista la revisión.

Revisión posterior

Cada uso del break-glass lo revisa alguien distinto de quien lo usó. La revisión no es una vista oral; plantea las preguntas de una revisión de diseño: ¿era una emergencia que el camino normal no podía atender?, ¿qué leyó o escribió la sesión?, ¿leyó columnas Restricted y era necesario?, y ¿habría podido una solicitud JIT cubrir la misma necesidad y, si es así, qué habría hecho que esa solicitud fuera lo bastante rápida?

La última pregunta es la que da fruto. Cada revisión confirma que el break-glass era la herramienta correcta o encuentra un hueco en el camino normal que lo convirtió en la única herramienta. Arreglar el hueco (un aprobador que falta, una ventana corta, una tabla que el techo no ofrece) reduce el siguiente uso. La revisión necesita además su propio reloj:

```
function checkBreakGlassReviews():
  for each r in breakGlassRequests(endedBefore: now()):
    review = reviewFor(r.id)
    if review is null and now() - r.expires_at > reviewDeadline:
      notify("security", "break-glass review overdue: " + r.id)
    if review is not null and review.reviewer == r.requester:
      notify("security", "break-glass reviewed by its own user:
" + r.id)
```

Una revisión que nadie hace es un control que ha dejado de existir sin que nadie lo note, así que su plazo dispara una alerta.

Cuando el gateway no está

Las personas llegan a la base de datos únicamente a través del access gateway (Capítulo 7), y tanto los grants JIT como los break-glass los concede el gateway a un login de sesión que él creó. Una caída del gateway es, por tanto, también una caída del break-glass. Es un trade-off del diseño, y el Capítulo 7 enuncia la misma decisión desde el lado del gateway.

La alternativa es una segunda puerta: un login de emergencia fuera del gateway. Resulta atractiva la noche en que se necesita y peligrosa todas las demás noches, porque una credencial que funciona sin el gateway funciona sin su logging, su identidad y su vencimiento; es la contraseña de administración compartida con mejor nombre.

Prefiero hacer que el gateway sea difícil de perder antes que hacer que su ausencia sea tolerable debilitando todo lo demás: más de una instancia, pocas dependencias (el registro de auditoría, los registros de solicitudes), un fallo ensayado y una señal de salud que el on-call vea primero. Si todas las instancias caen a la vez, no queda ninguna ruta humana a los datos, y no inventaría una aquí. El instrumento sellado del Capítulo 3 existe para trabajo a nivel de motor, necesita una ruta de red que el Capítulo 7 cierra a las personas, y se salta las protecciones `append-only` y `Restricted`, así que no es una ruta de acceso a datos. El Capítulo 15 vuelve sobre qué hacer en ese caso.

La elección no es gratuita. Un equipo con un gateway frágil puede querer razonablemente una segunda ruta mientras madura. Entonces hay que dejar por escrito qué es, quién puede obtener su credencial, cómo alerta su uso y qué la retira: una ruta con responsable y fecha de fin es una decisión, y una sin ellos es un resto olvidado.

Quando el motor no tiene grants a nivel de columna

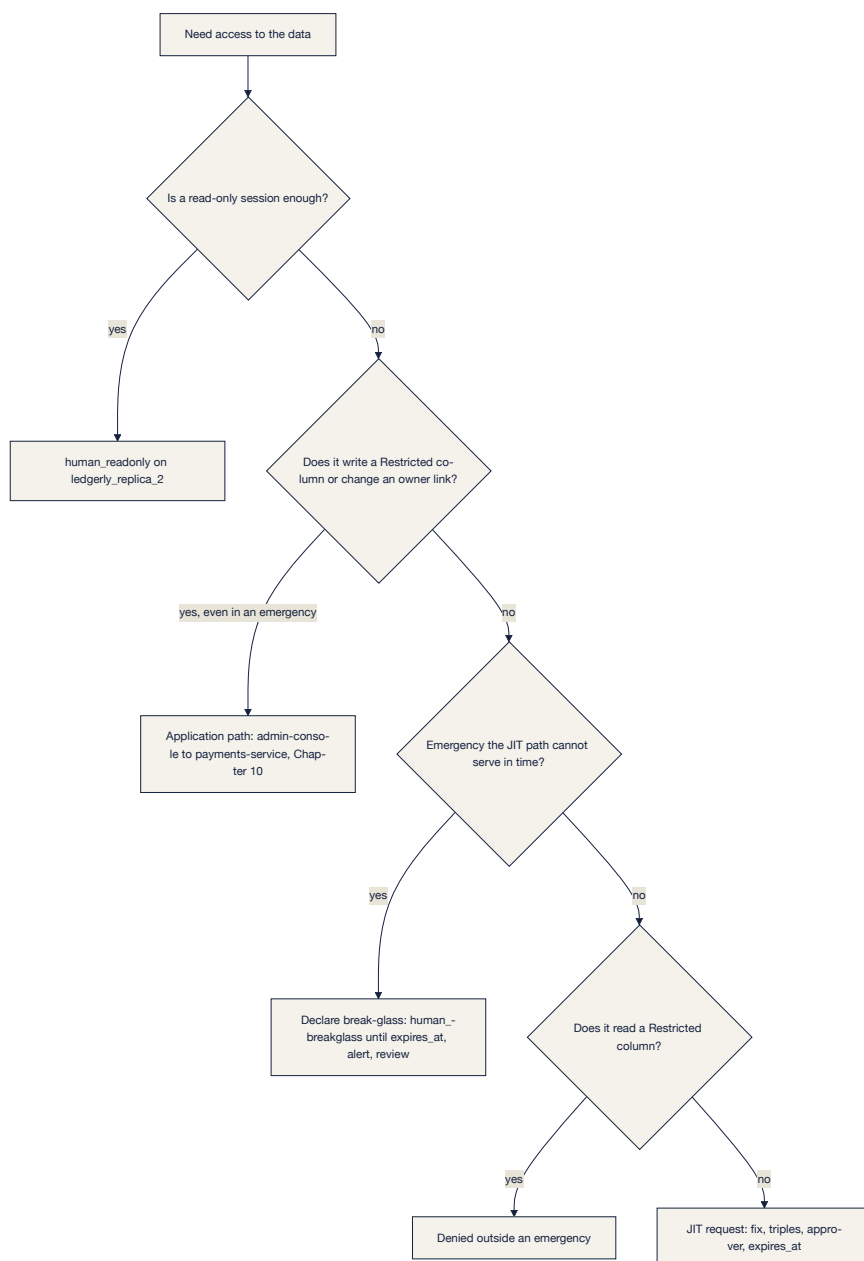
Todo lo anterior supone que el motor puede conceder un privilegio sobre columnas con nombre. El Capítulo 3 introdujo el único fallback que usa el libro para los motores que no pueden: las columnas `Restricted` viven en tablas separadas, `bank_account_secrets` y `payment_token_values`, fuera del techo y accesibles solo para `app_payments_service` y `human_breakglass`. Las tablas principales ya no contienen ninguna columna `Restricted`, así que un grant JIT a nivel de tabla sobre `bank_accounts` o `payment_tokens` no puede alcanzar ninguna. Una solicitud así no tiene lista de columnas: `columns` es `null`, y la tripleta dice “UPDATE sobre `payment_tokens`”.

El precio lo paga quien aprueba, que ha aprobado todas las columnas no `Restricted` de la tabla, incluidas las `Confidential`. Si quien solicita quería cambiar `expires_on`, quien aprueba está aprobando también `provider` y `last_four`. La pantalla de aprobación debe decir “todas las columnas no `Restricted` de esta tabla” con palabras, y la sentencia que lee quien aprueba carga con más peso. Un grupo de columnas que un equipo espera abrir a menudo merece su propia tabla pequeña.

El fallback no cambia las reglas para los datos Restricted. Una persona sigue sin escribirlos nunca con SQL en bruto; el cambio pasa por `admin-console` y `payments-service` después de que lo apruebe una segunda persona (Capítulo 10).

Elegir el camino

Una persona a las tres de la madrugada necesita saber qué puerta usar, y la respuesta pertenece al runbook como un flujo breve. El diagrama de abajo es su forma. Fíjate en los dos extremos que ninguna ruta humana a los datos atiende: una escritura en una columna Restricted, que va a la ruta de producto incluso en una emergencia, y una lectura de una de ellas fuera de una emergencia, que se deniega. El diseño lo dice en lugar de dejar la decisión a quien esté bajo presión.



Ledgerly a las 3 a. m.

Volvamos a Robin. Las filas atascadas se contaron en la sesión rutinaria, y el número coincide con el del runbook. La escritura es la solicitud vinculada a `fix-0093`, por una hora, citando la alerta abierta.

`chooseApproverPool` selecciona al on-call secundario y a la persona que

lidera el equipo, en ese orden, y el page va a la primera. El Ledgerly de antes habría abierto aquí la contraseña compartida. El nuevo tiene una espera, y la espera es el diseño funcionando.

Si nadie responde, el runbook plantea la pregunta que lo decide todo: ¿es esta una emergencia que el pool de aprobadores no puede atender a tiempo? Los recordatorios atascados pueden esperar hasta la mañana, así que Robin deja una nota en la alerta y se detiene, un desenlace que la contraseña de antes nunca permitía.

Supongamos, en cambio, que `payments-service` estuviera fallando, que el cobro de pagos se hubiera detenido para todos los clientes y que el diagnóstico necesitara una columna `Restricted` que ninguna solicitud JIT puede contener. Ese es un caso para el `break-glass`. Robin declara con la alerta como motivo; el gateway concede `human_breakglass` durante su ventana fija; se avisa a los contactos de seguridad y al equipo de plataforma. A la mañana siguiente alguien que no es Robin lee el log y plantea las preguntas de la revisión, una de las cuales es por qué el diagnóstico necesitaba una columna `Restricted`. La respuesta podría conducir a una vista de diagnóstico más segura el próximo trimestre.

Trade-offs y costos

La fricción es el precio, y hay que gastarla donde compre algo. Una solicitud JIT le cuesta a quien solicita un minuto de escritura y a quien aprueba un minuto de lectura, en el mejor de los casos, y una persona despertada en el peor. Para una tarea que ocurre dos veces por trimestre, es barato. Para una tarea que ocurre dos veces al día, es una funcionalidad de producto con disfraz operativo, y el Capítulo 10 describe cómo llevarla a `admin-console`.

Vincular a una corrección cuesta preparación. La corrección tiene que estar escrita antes de poder presentar la solicitud, lo que es más lento que un grant a secas. Es el precio de aprobar lo que se va a ejecutar en lugar de lo que podría ejecutarse.

La fatiga de aprobación es un fallo del diseño, no de quien aprueba. Una aprobación que llega en segundos, siempre, de la misma persona es teatro: o las solicitudes son triviales y `human_readonly` debería cubrirlas, o quien aprueba ha dejado de leer. El remedio son solicitudes más peque-

ñas, una pantalla que muestre en qué se convertirán los grants, aprobadores que conozcan las tablas y un pool lo bastante grande como para que nadie sea la persona por defecto. Dos señales pertenecen a la revisión de accesos: con qué frecuencia se rechazan o se devuelven las solicitudes, y con qué frecuencia responde la misma persona. Una tasa de rechazo de cero es tan probable que signifique que nadie mira como que las solicitudes son buenas.

El on-call carga primero con el costo. Se despierta a quien aprueba, quien solicita espera, y el equipo aprende un runbook que rara vez usa. Haz del pool de aprobadores una rotación, publícalo y asegúrate de que quienes lo integran tienen el contexto para leer una solicitud sobre `invoices` o `payments`.

El gateway es una dependencia de cada escritura. El job de revocación, el registro de auditoría y los registros de solicitudes son todas dependencias de un grant; cuando uno cae, los grants se detienen, lo que limita la exposición y ralentiza la respuesta a incidentes. El Capítulo 7 describe cómo reducir las probabilidades.

La conciliación produce ruido. La comprobación de drift y la de atrasos señalarán casos legítimos, como una revocación que corre con un minuto de retraso. Ajusta el período de gracia para que una señal signifique algo; una verificación silenciada es peor que ninguna, porque el equipo cree que está cubierto.

El break-glass tiene un costo social. Una persona que declara uno le está diciendo a los contactos de seguridad que el camino normal falló. Si la revisión parece un juicio, la gente dudará en emergencias reales; si es una revisión de diseño, no lo hará. La revisión sin culpas es lo que mantiene utilizable el rol.

Cuándo no usar esto. Un equipo de cuatro personas con una base de datos de bajo riesgo y sin datos Restricted no necesita pools de aprobadores ni un job de conciliación el primer día. Un proceso manual con un motivo, un límite de tiempo, una persona que confirma y un registro puede bastar durante un tiempo, si los grants se retiran cuando se acaba el tiempo y alguien lo comprueba. Construye el patrón en el orden de sus fallos: primero el vencimiento, luego el registro, luego el aprobador, luego la alerta.

Errores comunes

- **Conceder el rol del techo a un login “solo para esta tarea”.** Entonces cada sesión JIT tiene el techo completo y el alcance de la solicitud es decorativo. Concede las tripletas al login de sesión y mantén `human_jit_write` como un techo al que ningún login pertenece; la comprobación de drift señala a un miembro.
- **Aprobar las tripletas y no la sentencia.** Una lista de columnas es corta y fácil de leer, y aprobarla es aprobar una capacidad. Vincula la solicitud de escritura a una corrección revisada, deriva de ella las tripletas y toma `approved_hash` del registro de la corrección.
- **Revocar a partir de una lista aparte.** Una lista de tablas escrita a mano acabará por dejar fuera una tabla que concedió la solicitud. Deriva la revocación de `access_request_privilege`, el registro que impulsó el grant.
- **Confiar solo en el job de revocación.** Un job que ha corrido cada noche durante un año es fácil de dejar de vigilar. Dale a la sesión su propia hora de fin y ejecuta una comprobación independiente que compare los grants con las solicitudes.
- **Escribir la entrada de auditoría después del grant.** Si el log está caído, existe un grant que nadie puede ver. Escribe el registro, luego la entrada del log y luego los grants, y deja que una entrada de log fallida detenga el grant.
- **Aprobación automática de las solicitudes del on-call.** Las lecturas no necesitan solicitud y una escritura necesita una segunda persona. Acorta la espera con un buen pool de aprobadores.
- **Hacer el break-glass lo bastante largo como para estar cómodo.** Una ventana que nunca vence dentro de una jornada laboral se convierte en un rol de trabajo. Fija el máximo en el gateway y deja que una segunda declaración vuelva a alertar.
- **Dejar el conjunto de logging sin definir.** Quien revisa no puede juzgar una sesión cuyo registro nunca se especificó. Nombra el conjunto antes del primer uso.
- **Saltarse la revisión porque no pasó nada malo.** Las revisiones son donde se repara el camino normal. Ponle un plazo a la revisión y dispara una alerta cuando pase.

- **Construir una segunda puerta para el mal día del gateway.** Un login de emergencia fuera del gateway se salta su identidad, su logging y su vencimiento. Invierte primero en disponibilidad; si una segunda ruta es inevitable, dale un responsable, una alerta y una fecha de fin.

Ideas clave

- El acceso permanente cuesta más los días en que no se usa, así que la pregunta útil es quién necesita acceso ahora, para qué tarea y cómo sabrá el sistema que la tarea ha terminado.
- `human_jit_write` es un techo y nunca se concede a un login; una solicitud de escritura nombra su corrección revisada (`fix_id`), las tripletas se derivan de la corrección, y quien aprueba aprueba la sentencia, cuyo hash (`approved_hash`) debe coincidir con el del runner.
- La revocación se deriva del mismo registro `access_request_privilege` que impulsó el grant, se hace cumplir mediante la propia hora de fin de la sesión, se respalda con un job de revocación y la comprueba una conciliación independiente que no confía en ninguno de los dos.
- Las escrituras necesitan four-eyes: un aprobador que no sea ni quien solicita ni el autor de la corrección, que responda en la pantalla del gateway después de una verificación multifactor reciente. No hay aprobación automática para el on-call.
- El break-glass es una sesión `human_breakglass` autodeclarada con una ventana máxima fija, un conjunto de logging definido, una alerta en la declaración y una revisión obligatoria por alguien distinto de quien lo usó. No es una ruta rutinaria ni una ruta de corrección de datos, y su único riesgo aceptado es, hoy, un `UPDATE` sin acotar y, en el estado objetivo, una vez que el gateway del Capítulo 9 haga cumplir un tope, filas equivocadas dentro del tope.
- Un diseño sin un camino rápido autorizado genera uno no autorizado, así que la ruta de emergencia se diseña, se registra y se ensaya, y su dependencia del gateway es un costo que se nombra.
- Donde el motor no puede conceder sobre columnas, el fallback de tablas de secrets mantiene las columnas Restricted fuera de toda solicitud, y quien aprueba aprueba tablas enteras de columnas no Restricted.

Un grant, incluso uno vinculado, dice qué puede tocar una sesión y por cuánto tiempo, no con cuánto cuidado se comprueba la sentencia que hay detrás. El Capítulo 9 se ocupa de eso: qué hace el gateway con una sentencia antes de ejecutarla. El Capítulo 10 pasa después al trabajo en sí: cómo ejecutar una corrección de datos de modo que cambien las filas correctas, no cambien las incorrectas y una segunda persona pueda ver por qué.

Capítulo 9 — Ejecución mediada: análisis de consultas y operaciones peligrosas

El equipo de soporte de Ledgerly tiene un asistente en su herramienta de chat. Sam le pide anular un lote de facturas duplicadas por un retry de facturación, y el asistente, que corre bajo su propia identidad, redacta una corrección: un `UPDATE` sobre `invoices` con una cláusula `WHERE` sobre la etiqueta de lote del retry. Jordan, del equipo de datos, revisa la sentencia y la aprueba. La sentencia es SQL válido, el grant la cubre, quien aprueba la leyó, y la etiqueta era correcta cuando Jordan la comprobó.

Aun así, la sentencia coincide con más de las doce filas que nombraba la corrección. Un segundo job había reutilizado la etiqueta de lote dos días antes, y nada en el texto de la sentencia lo dice. Nadie fue descuidado: Sam describió la tarea con exactitud, el asistente escribió lo que se le pidió y Jordan revisó lo que se le mostró. Lo que faltaba era un paso entre “aprobado” y “confirmado”, un lugar donde algo mire lo que la sentencia va a hacerle a los datos y no solo lo que dice.

Los capítulos 3 a 8 decidieron quién puede tener un grant, por qué ruta y por cuánto tiempo. Este capítulo añade la parte Ejecución del modelo de cinco partes: el paso en el que el gateway se ocupa de cada sentencia, en lugar de entregar una sesión y confiar en quien la tiene. La pregunta que responde es esta: ¿qué debería interponerse entre una sentencia aprobada y los datos, cuando la aprobación por sí sola no puede decir qué va a hacer la sentencia?

El problema y por qué es difícil

Un session handout le da a una persona un login y deja que su cliente hable directamente con la base de datos. Todo lo que ocurre después es asunto del cliente: lo que se escribe, cuántas filas toca, si confirma en el

acto. El grant acota qué tablas y columnas son alcanzables, y nada acota lo que hace una sentencia concreta dentro de ese alcance. En el estado inicial, un cliente de escritorio envía lo que se teclea y lo confirma de inmediato.

La ejecución mediada invierte el arreglo. La sentencia va a un componente que no es el cliente de la persona, y ese componente decide si se ejecuta, con qué límites y cuándo se confirma. El gateway del Capítulo 7 ya está en la ruta de los datos, así que es el lugar natural para esto. La persona sigue escribiendo la sentencia, o lo hace una automatización; lo que cambia es que ninguna sentencia llega a los datos sin pasar por el bucle de la sección siguiente.

Es un espectro, no un interruptor. La forma más barata es un session handout con unos pocos ajustes que el gateway hace cumplir, como un timeout de sentencia y un row cap. La siguiente forma comprueba cada sentencia antes de reenviarla. La forma más mediada aprueba cada sentencia individualmente y la ejecuta desde un worker, de modo que la persona nunca tiene una sesión de base de datos para la escritura. Algunas herramientas aprueban cada consulta en lugar de una ventana de tiempo. En el momento de escribir esto, QueryProxy, por ejemplo, hace que una segunda persona apruebe la sentencia y que un worker la ejecute, de modo que quien desarrolla nunca tiene la credencial de la base de datos. Eso cambia la velocidad de una sesión interactiva por una aprobación vinculada a la sentencia exacta, y otras herramientas de aprobación de consultas funcionan de manera parecida.

La mediación es difícil por tres razones. La primera es que SQL es expresivo: el mismo efecto se puede escribir de muchas maneras, y que una sentencia sea “peligrosa” depende a menudo de datos que el verificador no ha leído. La segunda es que cada comprobación tiene un costo en latencia, en rechazos falsos y en las personas que aprenden a esquivarla. La tercera es que un verificador en el que se confía demasiado se convierte en lo que nadie revisa. La posición de este capítulo es que el análisis de sentencias reduce errores y no es la frontera de autorización. La frontera son los grants. El trabajo del gateway es hacer que la sentencia equivocada más común sea difícil de ejecutar por accidente, y lograr que cada escritura lleve un límite que haga cumplir algo distinto de quien escribe.

El bucle del gateway para cada sentencia

Toda sentencia que envía una sesión, humana o no, pasa por el mismo bucle. El orden importa más que cualquier paso concreto, así que aquí está como pseudocódigo antes de explicar las piezas.

```
function handleStatement(session, statement):
    if statement.changesSetting(guardedSettings):
        return refuse(session, "guarded setting")
    verdict = evaluateStatement(session, statement)
    // the record comes first: a statement that cannot be recorded
    does not run
    try:
        recordIntent(session, statement, verdict)
    on RecordUnavailable:
        return refuse(session, "record unavailable")
    if verdict.action == "deny":
        return refuse(session, verdict.reason)
    if verdict.action == "gate":
        return routeToGate(session, statement, verdict) // approval,
or the product path
    if statement.isRead:
        return streamRows(session, statement, cap: session.rowCap)
// cursor closes at the cap
// auto-commit is off: the runner owns the transaction of a fix,
the gateway owns the rest
    tx = transactionFor(session)
    result = execute(tx, statement)
    tx.rowsSoFar = tx.rowsSoFar + result.rows // counted across
every statement of the fix
    if tx.rowsSoFar > verdict.bound:
        failTransaction(tx) // its owner can
only roll back
        return refuse(session, "more rows than approved")
    showRowCount(session, tx.rowsSoFar) // before COMMIT,
while undo is free
    if tx.ownedByGateway:
        if not countAccepted(session, tx.rowsSoFar, verdict.bound):
            rollback(tx)
            return refuse(session, "count not accepted")
        commit(tx) // the commit point
for a person or a migration
    recordOutcome(session, statement, result) // failure is
queued and alerts; a commit stands
    return result
```

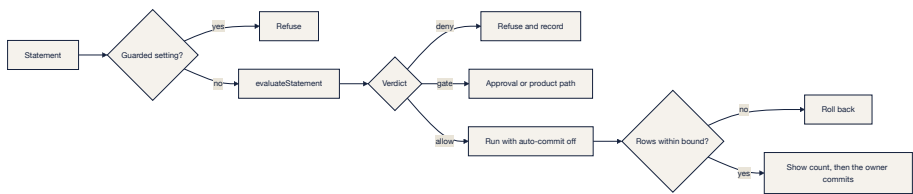
Guarded settings. El gateway abre cada sesión con sus límites aplicados: el timeout de sentencia, un valor por defecto de solo lectura, el rol y el límite de conexiones según el tipo de sesión (Capítulo 5). Un cliente puede enviar un `SET` que levante cualquiera de ellos, así que el gateway mantie-

ne una lista de guarded settings y rechaza una sentencia que cambie alguno. La palabra “no anulable” solo se gana donde hay un mecanismo que la hace cumplir. Un timeout fijado como valor por defecto del rol puede levantarlo la sesión a menos que el gateway rechace el SET o la plataforma lo fije, de modo que un límite que no tiene ninguna de las dos cosas es un valor por defecto y no una garantía. El costo es que algunos clientes envían esos ajustes al conectarse y dejan de funcionar hasta que alguien los incluye en la lista.

El row cap. En las lecturas, el gateway cuenta las filas a medida que las transmite y cierra el cursor al llegar al tope, se lo dice a la persona y registra el evento. El tope no dice que una persona solo pueda ver cierto número de filas al día, y una persona decidida puede recorrer una tabla por tramos. Dice que una sola consulta equivocada no arrastra una tabla entera a un laptop. Las extracciones legítimas grandes pertenecen a la ruta de export del Capítulo 13 y a la copia analítica enmascarada del Capítulo 12, donde el nivel decide quién aprueba.

Auto-commit desactivado para las escrituras. Una sesión de escritura ejecuta cada escritura dentro de una transacción explícita. Para una persona o una migración, la abre el gateway; para una corrección de datos, la abre el runner del Capítulo 10, y el gateway reenvía cada sentencia de la corrección a esa única transacción. En ambos casos el gateway cuenta las filas afectadas en todas las sentencias y las compara con el límite, mientras que la decisión de hacer COMMIT queda con el propietario de la transacción: un gateway que confirmara por su cuenta cada sentencia de una corrección rompería la transacción única del runner y haría que su dry run fuera un cambio real. Cuando el conteo supera el límite, el gateway rechaza la sentencia y marca la transacción como fallida, de modo que su propietario solo pueda hacer rollback. Es la forma barata de la reversibilidad del Capítulo 1: un update que debía tocar una fila y reporta cuatrocientas se detiene mientras deshacer todavía es gratis. En una sesión interactiva, “mostrado” significa que una persona ve el conteo y lo confirma. Para el runner y para la automatización, “mostrado” significa que el gateway compara el conteo con el límite que registró la aprobación y hace fallar la transacción ante cualquier exceso. El costo es el paso adicional y los bloqueos de la transacción, que se mantienen mientras la persona decide; una confirmación que nadie responde mantiene un bloqueo de fila, por lo que la confirmación tiene un timeout que hace rollback.

El registro primero. El gateway escribe la intención antes de que la sentencia se ejecute y el resultado después. No hay ninguna transacción que abarque el log del gateway y la base de datos, así que el orden es lo que sostiene la garantía. Si no se puede escribir la intención, la sentencia no se ejecuta. Si no se puede escribir el resultado después de un commit, el commit se mantiene, la entrada se pone en cola y se alerta al equipo de plataforma; un log que solo funciona cuando nada va mal no merece ese nombre. El Capítulo 14 explica dónde vive el registro.



Clasificar una sentencia

El gateway decide qué hacer con una sentencia en `evaluateStatement`. Clasifica la sentencia por lo que hace y dónde, consulta la matriz en la sección posterior a la siguiente y, para una escritura que la matriz permite con un límite, busca el límite y compara con él la estimación del planner. Cada rechazo y cada gate vienen de la matriz; la función no tiene retornos tempranos propios aparte de los del parser.

```

function evaluateStatement(session, statement):
    parsed = parse(statement)
    if parsed.failed or parsed.isMultiStatement:
        return deny("cannot be analyzed")           // fail closed
    // nine classes, one per matrix row; the most specific class wins
    class = classify(parsed)
    cell = matrix[class][session.kind]              // deny, gate,
allow or allow with a bound
    if cell.action != "allow":
        return cell                                // a deny, or a
gate with its route
    if class == "read":
        return allow(cap: session.rowCap)
    if cell.bound == "fix":
        binding = checkBinding(session, statement)  // the next
statement of the approved fix
        if binding.action == "deny":
            return binding
        bound = binding.bound
    else:
        bound = session.writeBound()                // the break-glass row cap, or
the backfill rule's limit
        if bound is none:
            return deny("no approved count")
        estimate = plannerEstimate(parsed)           // asks the
planner; runs nothing
        if estimate > bound * tolerance:
            return deny("estimate exceeds the approved count")
        return allow(bound: bound)

```

Hay tres decisiones que merece la pena defender. Primera, se rechaza una sentencia que el parser no puede leer. Si el gateway ejecutara lo que no pudo analizar, el análisis se aplicaría solo a las sentencias que nunca fueron un problema. Un cliente que envía lotes de sentencias en una sola cadena se rompe con esta regla, y eso es un costo que hay que nombrar en la migración; la respuesta es enviarlas de una en una.

Segunda, la clasificación mira el efecto y no el nombre de la persona ni de la herramienta. Un `DELETE` en una migración y un `DELETE` en una sesión JIT son la misma clase, y la matriz decide qué puede hacer con él cada lane. Tener un único clasificador con una sola tabla, y no un conjunto de reglas aparte por ruta, es lo que mantiene la política lo bastante pequeña como para revisarla.

Tercera, la estimación del planner se compara con el conteo esperado, pero solo como rechazo temprano. El planner estima a partir de estadísticas, y las estadísticas se quedan obsoletas; una estimación puede ser tan

fácilmente demasiado baja como demasiado alta. La estimación permite al gateway rechazar una sentencia claramente equivocada antes de que tome bloqueos. El control es el conteo real, comparado dentro de la transacción antes del `COMMIT`. La tolerancia es un parámetro que el equipo de plataforma fija y registra, y yo empezaría estrecho y la relajaría solo cuando se midan rechazos falsos, no cuando se supongan.

Lo que el análisis estático no puede ver

El texto de una sentencia dice poco de su efecto sobre las filas. Un parser puede ver que falta un `WHERE`, que un predicado es siempre verdadero, que una sentencia toca una tabla que no debería o que llama a una función. No puede ver cuántas filas coincide hoy un predicado, que es el problema de la apertura: un filtro válido, sobre una etiqueta que ahora coincide con más filas de las que nadie esperaba.

Hay otros casos que quedan fuera de lo que el análisis puede resolver. Un filtro puede ser válido y aun así amplio, y coincidir con la mayor parte de una tabla por los datos y no por la sintaxis. Un join puede multiplicar filas. Una vista o una función pueden esconder una escritura o una lectura más amplia detrás de un nombre de aspecto inofensivo, y el gateway no puede ver dentro del cuerpo de una función a la que el rol tiene permitido llamar. Un efecto secundario desencadenado ocurre después de la sentencia y es invisible en su texto. Y una sentencia puede tener exactamente la forma correcta y el significado equivocado: las filas correctas según el filtro, las filas equivocadas según la intención, algo que ninguna herramienta puede juzgar.

Las herramientas de código abierto como QueryProxy comprueban las sentencias al enviarlas, rechazan un `UPDATE` o `DELETE` cuyo `WHERE` falta o es siempre verdadero y limitan los `SELECT` sin acotar; en el momento de escribir esto, su propia documentación describe este parser como una capa de defensa y no como la frontera de autorización, que es el punto de esta sección, y otros gateways y herramientas de aprobación adoptan enfoques parecidos.

De esto se sigue un reparto del trabajo. El análisis detecta los errores comunes de forma barata y temprana. Los grants (Capítulo 3) fijan qué es alcanzable en absoluto, de modo que una sentencia que se escapa sigue

acotada por el rol. La comprobación del conteo en la transacción detecta el caso en que los datos discreparon de quien escribió. La aprobación, con una segunda persona que lee la sentencia y el conteo esperado, detecta, algunas veces, el caso en que la forma era correcta y la intención equivocada. Ninguna de estas cosas es completa, y el diseño se apoya en su solapamiento y no reclama ninguna de ellas como la frontera.

El mismo límite da forma al nivel de export del Capítulo 13. El gateway calcula el nivel de un export a partir de las columnas de origen que lee la consulta, no a partir de los nombres de las columnas del resultado, y un origen que no puede resolver cuenta como el nivel exportable más alto; donde ni siquiera eso se puede decidir, el export se rechaza. Es la misma regla de fail closed que la del parser: lo que el análisis no puede leer, no lo deja pasar.

La matriz deny, gate y allow

Cada clase de sentencia recibe una de tres respuestas en cada lane. **Deny** significa que el gateway rechaza y registra el rechazo. **Gate** significa que la sentencia no se ejecuta tal como se envió: necesita una aprobación que todavía no tiene, o pertenece a otra ruta. **Allow** significa que se ejecuta bajo los límites del bucle anterior; donde una celda dice “bound”, la escritura se permite únicamente dentro de un límite que viene de la fuente que nombra la celda, y el conteo real se compara con él antes del COMMIT. Las columnas son los lanes que ha descrito el libro, y las filas son las clases de sentencia que produce el clasificador.

Clase de sentencia	Sesión de lectura humana	Escritura JIT	Break-glass	Runner de correcciones de datos	Migración
Lectura, dentro del row cap	Allow	Allow	Allow	Allow	Allow
Escritura acotada, dentro del conteo aprobado	Deny	Allow, vinculada a la corrección (solo runner)	Allow, vinculada a un row cap (nota obligatoria)	Allow, vinculada al hash y al conteo aprobados	Allow, vinculada a la regla de backfill

Clase de sentencia	Sesión de lectura humana	Escritura JIT	Break-glass	Runner de correcciones de datos	Migración
Escritura sin acotar	Deny	Deny	Deny	Deny	Gate: aprobador designado
Cambio de vínculo con el propietario	Deny	Gate: ruta de producto	Gate: ruta de producto	Gate: ruta de producto	Gate: aprobador designado
Escritura en una columna Restricted	Deny	Gate: ruta de producto	Gate: ruta de producto	Gate: ruta de producto	Gate: aprobador designado
Export	Gate: por nivel	Deny	Deny	Deny	Gate: aprobador designado
Cambio de esquema (DDL)	Deny	Deny	Deny	Deny	Allow: lane revisado
Cambio de privilegios	Deny	Deny	Deny	Deny	Gate: revisión de plataforma
Escritura en ledger_entries o audit_events	Deny	Deny	Deny	Deny	Gate: revisión de plataforma

Leerla fila por fila muestra el diseño. La sesión de lectura humana no tiene ruta de escritura, así que toda clase de escritura se deniega ahí, y el único gate es el nivel del export. Una sesión de escritura JIT reenvía las escrituras solo desde el runner, así que su celda de escritura acotada es un allow con un vínculo y no una puerta abierta. El break-glass es el único lane donde una persona escribe directamente, y por eso su escritura acotada se permite solo bajo un row cap y con una nota, y por eso el tamaño de ese tope es la decisión de dimensionamiento más difícil del libro: una emergencia no es la tarea planificada, así que ningún tope se puede elegir a partir de la tarea. El tope debe fijarse a partir de lo que el rol break-glass podría necesitar razonablemente reparar y revisarse después de cada uso. Una escritura sin acotar se deniega ahí. Lo que queda es una escritura que se mantiene dentro del tope y afecta a las filas equivocadas, un riesgo

aceptado que la revisión existe para dejar al descubierto, como dice el Capítulo 8; el Capítulo 8 dice también qué se sostiene hoy, antes de que un gateway haga cumplir el tope.

El cambio de vínculo con el propietario merece su propia fila. Un `UPDATE` de `customer_id` en `bank_accounts` o `payment_tokens` pasa el techo JIT, porque la columna vincula una fila con su propietario y no es `Restricted`. Aun así, mover la fila a otro cliente cambia de quién es esa cuenta de pago o ese token de pago. Por eso el clasificador lo nombra como una clase propia, y todos los lanes salvo una migración responden con un gate que lo envía a la ruta de producto: `admin-console` se lo pide a `payments-service`, una segunda persona aprueba y el cambio se registra en `audit_events`. Una migración en el lane normal lo somete a gate para un aprobador designado, como hace el Capítulo 11 con cualquier sentencia marcada, y el lane expedited lo rechaza; un cambio de esquema no tiene nada que hacer reasignando la propiedad, y si alguna vez hace falta una reasignación masiva, es una funcionalidad de producto que construir y no una sentencia que ejecutar. Las escrituras en columnas `Restricted` (`iban`, `account_holder_name` y `token_value`) siguen la misma ruta, y por eso ningún lane permite a una persona escribirlas con SQL en bruto. El único caso de migración es mover las columnas `Restricted` a las tablas de secrets separadas del Capítulo 11, que va a un aprobador designado y no a un rechazo.

La columna de migración coincide con lo que describe el Capítulo 11. Su clasificador marca las sentencias para un aprobador designado o para una revisión de plataforma, y rechaza solo en el lane expedited. En esta tabla las marcas aparecen como gates donde una persona tiene que decir que sí, y los allows son el cambio de esquema revisado y el backfill que se queda dentro de su regla. Las tablas `append-only` son igual: una migración que las toca va al equipo de plataforma, y los demás lanes lo deniegan. Un segundo conjunto de reglas para sentencias prohibidas sería una segunda política que mantener sincronizada con esta, así que el pipeline de migraciones reutiliza el clasificador y la matriz y añade únicamente sus reglas de lane.

La matriz es deliberadamente pequeña. Nueve clases y cinco lanes caben en una página, y una política que cabe en una página se lee. Cada clase o lane nuevo que se le añade es una decisión que alguien tiene que defen-

der, y el fallo habitual es una matriz que acumula una excepción por incidente hasta que nadie sabe decir qué prohíbe. La mantendría en control de versiones, revisada como código, con el equipo de plataforma como responsable y el equipo de datos como revisor obligatorio de las filas de escritura.

Vincular la aprobación a la sentencia

Una aprobación que cubre “una escritura sobre `invoices`” deja pasar cualquier escritura sobre `invoices`. La aprobación vinculada a la sentencia significa que la aprobación nombra esta sentencia, y el gateway rechaza cualquier otra. El Capítulo 8 describe la solicitud y el Capítulo 10 el runner; esta sección es la regla que los conecta, enunciada una sola vez.

Una solicitud de escritura lleva un `fix_id` y un `approved_hash`. El hash es el hash de las sentencias que se revisaron, tomado sobre el conjunto entero cuando una corrección tiene más de una, y su origen importa más que la comprobación. Proviene del registro de merge de la corrección, o del registro de la solicitud aprobada, y nunca de la entrada del runner. Si el runner aportara su propio hash, el runner estaría comprobando su propio trabajo, y un cambio en la sentencia llegaría junto con un hash que coincide. Un hash que se origina en el registro que aprobó una segunda persona es un hash que avaló alguien distinto de quien escribe.

```
function checkBinding(session, statement):
  request = loadApprovedRequest(session.requestId)    // from the
  store, not from the caller
  if request is none or request.status != "approved":
    return deny("no approved request")
  if request.fixId is none:
    return deny("no fix named")                      // a SELECT-
only request has no fix and no hash
  fix = loadMergeRecord(request.fixId)                // the
statements, read from the record
  if hashOf(fix.statements) != request.approvedHash:
    return deny("record differs from the approved hash")
  next = fix.statements[session.transaction.statementCount]
  if normalize(statement) != normalize(next):
    return deny("statement differs from the approved one")
  return allow(bound: fix.expectedRows)              // the count
comes from the hashed record
```

El gateway comprueba una sentencia cada vez, pero el hash cubre el conjunto: carga las sentencias del registro de merge, confirma que su hash es el aprobado y acepta únicamente la siguiente sentencia en ese orden. El límite es el conteo esperado de filas del mismo registro, y no un número que lleve la solicitud, de modo que el hash protege también el conteo. El commit queda en la transacción del runner, así que una corrección con varias sentencias se aplica entera o no se aplica.

Dos reglas completan el vínculo. En una sesión de escritura JIT, el gateway reenvía las escrituras únicamente desde la conexión del runner. El cliente de la propia persona todavía puede leer dentro de las tripletas concedidas, pero una escritura enviada desde él se rechaza, porque de otro modo el hash sería una comprobación que un cliente podría rodear. Una protección es un mecanismo solo si la sesión no puede escribir de ninguna otra manera. La segunda regla es el caso inverso: una solicitud JIT que contiene solo tripletas `SELECT` no tiene corrección ni hash, así que no hay nada a lo que vincular una escritura, y la escritura se deniega. Una solicitud de lectura no puede convertirse en una de escritura enviando una sentencia diferente.

Donde el gateway solo intermedia una credencial y el cliente se conecta directamente, nada de esto es exigible, y el libro lo dice sin rodeos: el hash entonces registra una intención y no previene nada. El vínculo es una propiedad de la ruta de los datos y existe solo donde el gateway está en ella.

El caso del runbook es el que tiene una plantilla. El registro de merge de un runbook guarda el hash de la plantilla de la sentencia, y la solicitud lleva solo la lista de claves. El runner sustituye las claves en la plantilla almacenada. Para este caso el gateway comprueba el hash contra la plantilla y no contra el texto sustituido: confirma que la sentencia enviada es la plantilla aprobada con solo la lista de claves rellena, y usa el número de claves como límite, junto con las tripletas de la solicitud. El gateway nunca toma una sentencia ni un hash de quien llama y quiere que se haga el trabajo.

Quienes llaman y no son personas

Los jobs, los scripts de runbook, los bots de ChatOps y los agentes de IA envían sentencias, y ninguno es una persona. La regla de este capítulo es que pasan por el mismo bucle, como principales propios. No hay un ca-

mino más ligero para la automatización, porque la automatización es donde un error se repite a velocidad de máquina.

Quien llama de forma automatizada tiene una workload identity de la plataforma y no la sesión de una persona, y un valor por defecto de solo lectura. Abre su sesión bajo su propio principal, de modo que sus lecturas están sujetas al mismo row cap, timeout y guarded settings que las de cualquiera. Si quiere escribir, sigue la misma ruta que una persona: nombra una corrección, la aprueba un aprobador que es una persona distinta de la que activó la automatización, y sus escrituras pasan por el runner con un conteo esperado. La regla de las dos personas del Capítulo 8 se aplica aquí también: quien le pidió a la automatización que redactara la corrección no puede ser quien la apruebe, y no hay aprobación automática por parte de un cliente que no sea una persona. Un agente que propone una sentencia la está proponiendo, y la aprobación es una persona que la lee.

El registro nombra a las dos partes: la automatización que envió la sentencia y la persona que la activó. Esta es la mitad de la rendición de cuentas. Un registro que dijera solo “lo hizo el asistente” terminaría el rastro en una máquina, y uno que dijera solo “lo pidió Sam” ocultaría qué herramienta convirtió la solicitud en SQL. Se guardan ambos nombres, y la revisión de accesos pregunta por los dos.

Un agente de IA añade una preocupación que los demás clientes no tienen: sus sentencias se generan a partir de texto, parte del cual no escribió él. Un ticket, una línea de log o un mensaje de un cliente pueden acabar en el contexto que produce la sentencia. La defensa no es leer el texto con más cuidado. Es que los privilegios del agente no cambian con lo que lee. Sus grants son los mismos, sus escrituras siguen necesitando la aprobación de una persona sobre la sentencia exacta, y una sentencia que llega por una ruta que el agente no debería tener, como una escritura fuera del runner, la rechaza la misma matriz. Sea lo que sea que se persuadiera al agente de proponer, no puede aprobarlo ni ejecutarlo sin el límite.

Para un agente de solo lectura, los límites útiles son los ordinarios: un row cap estrecho, un timeout corto, una vista o una copia enmascarada en lugar de las columnas Restricted, y un presupuesto por principal, de modo que un bucle que sale mal agote su propia asignación y no la compartida. Para un agente que escribe, el límite útil es que sus escrituras sean pocas y pequeñas, porque cada una necesita a una persona. Si eso hace que el

agente sea demasiado lento para ser útil en escrituras, la conclusión correcta es que la tarea debería convertirse en una funcionalidad de producto a la que llame el agente, no que la aprobación deba relajarse.

Trade-offs y costos

La mediación cuesta tiempo. Cada sentencia pasa por un componente más, y una escritura sometida a gate espera a una persona. En las lecturas interactivas sobre una réplica, la latencia añadida es pequeña y en su mayor parte invisible. En las escrituras es el objetivo: el retraso es donde ocurre la segunda mirada. No lo eliminaría en los casos urgentes; haría que el gate del lane break-glass sea rápido y ruidoso, y lo revisaría después.

Cuesta compatibilidad. Un gateway que analiza sentencias y rechaza lo que no puede analizar romperá algunos clientes, algunos drivers y algunos hábitos. Rechazar sentencias `SET`, cadenas con varias sentencias y sintaxis inusual produce una lista de cosas que dejan de funcionar. La lista es más corta si se recoge antes del despliegue, a partir de los registros de sentencias de las sesiones existentes, que si se descubre a través de tickets de soporte. Presupuesta un período en el que el gateway registre sus veredictos pero rechace solo los casos más claros, y pasa a hacerlo cumplir clase por clase a medida que bajen los rechazos falsos.

Cuesta un componente que debe estar disponible. La comprobación de sentencias está en la ruta de cada escritura, y el Capítulo 7 describe por qué ese componente debe mantenerse pequeño. Cuando la comprobación está caída, el valor por defecto que acota el daño es que las escrituras se detienen y las lecturas sobre la réplica continúan. Una comprobación fail-open le enseña al equipo que la comprobación es opcional, y la primera caída termina con la política.

Cuesta una política que mantener. La matriz, la lista de guarded settings, la tolerancia y los row caps son parámetros que alguien tiene que poseer. Una política sin responsable deriva hacia lo que necesitó el último incidente. La matriz es el artefacto que hay que mantener pequeño, revisado y versionado.

Hay además un costo que no se puede pagar con ingeniería: el análisis da una falsa sensación de seguridad. Un equipo que tiene un parser que rechaza `UPDATE` sin `WHERE` puede empezar a tratar una sentencia que pasó

como una sentencia que no puede hacer daño. La matriz dice lo que el gateway rechaza, y la apertura de este capítulo muestra lo que todavía puede hacer una sentencia que pasa. La comprobación del conteo y la segunda persona existen para ese caso, y no deberían reducirse porque el parser sea bueno.

Errores comunes

Tratar el analizador como la frontera. El analizador es una capa que detecta la sentencia equivocada más común. Los grants deciden qué es alcanzable, y un diseño que deja que el analizador cargue con ese peso falla la primera vez que una sentencia es válida y equivocada.

Dejar que el runner o quien llama aporte el hash. Un hash que llega con la sentencia solo demuestra que la sentencia coincide consigo misma. El hash viene del registro de merge o de la solicitud aprobada, y la comprobación corre en el gateway, donde quien llama no puede editarla.

Dejar el auto-commit activado en las sesiones de escritura. Con el auto-commit activado, el conteo de filas es un informe a posteriori. Desactivarlo y mostrar el conteo antes del COMMIT es lo que hace barato un conteo equivocado.

Fallar en abierto cuando el parser no puede leer una sentencia. Ejecutar lo que no se pudo analizar hace que el análisis cubra solo los casos inofensivos. Recházala, y mantén una lista corta de los clientes que necesitan cambios.

Un camino más ligero para la automatización. Darles a los jobs y a los agentes un atajo alrededor del bucle, porque son de confianza o porque tienen prisa, quita los controles donde el volumen es mayor. La automatización pasa por el mismo bucle bajo su propio principal.

Una matriz que acumula una excepción por incidente. Cada excepción es una decisión sin responsable. Mantén la tabla en las clases y lanes que existen, y cuando un caso no encaja, cambia el diseño o construye la funcionalidad.

Leer un row cap como un límite de la exposición. Un tope acota una consulta, no lo que una persona puede leer en un día. Si la preocupación es la exposición, pertenece al trabajo de enmascaramiento y de niveles del

Ideas clave

- La ejecución es la manera en que una sentencia llega a los datos: quién la envía, si algo la inspecciona antes, qué acota su efecto y cómo se confirma. Los grants no responden a estas preguntas.
- La ejecución mediada pone un componente entre la sentencia y los datos. La forma más barata es un puñado de ajustes que se hacen cumplir; la más fuerte aprueba cada sentencia y la ejecuta desde un worker.
- Toda sentencia, de una persona o de una automatización, pasa por un solo bucle: guarded settings, evaluación, registro primero, un row cap para las lecturas y auto-commit desactivado para las escrituras con el conteo de filas mostrado antes del COMMIT.
- El análisis reduce errores y no es la frontera de autorización. La frontera son los grants, y el conteo real de filas comprobado dentro de la transacción detecta lo que el parser y el planner no pueden.
- Una matriz pequeña de clase de sentencia por lane da cada respuesta como deny, gate o allow. El cambio de vínculo con el propietario y las escrituras en columnas Restricted pasan por gate y se envían a la ruta de producto.
- La aprobación está vinculada a la sentencia: el `approved_hash` viene del registro de merge o de la solicitud aprobada, nunca del runner, y en una sesión de escritura JIT el gateway reenvía las escrituras únicamente desde el runner.
- Los jobs, los runbooks, ChatOps y los agentes de IA son principales con identidad propia, un valor por defecto de solo lectura y el mismo bucle. Sus registros nombran tanto a la automatización como a la persona que la activó.

El bucle y la matriz dicen qué hace el gateway con una sentencia. No dicen cómo escribir una corrección que valga la pena aprobar: cómo demostrar el conteo, conservar un snapshot, planificar el rollback y lograr que una segunda persona la revise. El Capítulo 10 se ocupa de ese trabajo.

Capítulo 10 — Consultas ad hoc y correcciones de datos

Imagina un jueves por la tarde en Ledgerly. Un bug en `invoice-worker` ya se corrigió y se desplegó, pero mientras estuvo en producción dejó un lote de facturas en el estado `sending`, aunque los correos ya habían salido. Soporte pregunta por ellas, los recordatorios de los clientes están en pausa, y Robin, la persona de backend que está de on-call, ve la forma de la reparación en una sola línea: devolver el estado a `sent` en las facturas afectadas y nada más.

Robin abre una sesión a través del access gateway. La sesión por defecto es de solo lectura, así que esa línea no puede ejecutarse. Robin necesita pedir acceso de escritura, y esa solicitud tiene que nombrar tablas y columnas. Alguien más tiene que mirar la sentencia antes de que toque el primary. Mientras tanto el reloj corre, y la reparación es tan pequeña que el proceso que la rodea parece más grande que el trabajo.

Este capítulo añade un paso del estado objetivo: las escrituras se hacen mediante un procedimiento revisado de corrección de datos y grants just-in-time dentro del techo de `human_jit_write`. El Capítulo 7 dio a las personas una sola puerta, y el Capítulo 8 construyó los grants que la abren por un tiempo limitado. Aquí preguntamos qué debe hacer una persona una vez que la puerta está abierta: cómo leer, cómo escribir y cómo hacer que el segundo tipo de trabajo sea lo bastante seguro como para hacerlo con calma un jueves por la tarde.

En el modelo de cinco partes del Capítulo 1, este capítulo trata de la **Ejecución**. Robin ya tiene todas las credenciales que la reparación necesita, así que el riesgo no está en quién tiene una credencial. Está en lo que autoriza el grant, en lo que hará una sentencia cuando se ejecute, en quién la leyó primero y en qué evidencia queda. La pregunta del capítulo es esta:

¿cuál es la forma más pequeña y más revisable de que una persona cambie datos de producción, y cuándo debe dejar el cambio de ser trabajo de una persona?

El problema y por qué es difícil

Todo sistema que guarda datos reales llega con el tiempo a un estado que su propio código no puede reparar. Un worker se cayó entre dos pasos. Una migración dejó una columna a medio llenar. Un cliente se fusionó con otro por error. La aplicación no tiene un endpoint para “arreglar esto”, porque nadie esperaba necesitarlo, y escribirlo llevaría más tiempo que la reparación. Así que una persona escribe una sentencia y la ejecuta contra producción. Eso es una **corrección de datos**: un cambio en datos de producción hecho por una persona fuera de la ruta normal de la aplicación, para reparar un estado que la aplicación no puede reparar.

La primera dificultad es que la herramienta peligrosa y la herramienta útil son la misma. Una sesión SQL que puede responder “¿cuántas facturas están atascadas?” puede, con una palabra más, cambiarlas todas. En el estado inicial del Capítulo 1, el cliente GUI tiene el login de administrador compartido, hace auto-commit de cada sentencia y ejecuta el texto que esté resaltado. La distancia entre una pregunta y un cambio irreversible es una palabra.

La segunda dificultad es que la persona que hace el trabajo suele estar bajo presión y suele tener razón. Un proceso que trata cada corrección como una amenaza enseña a la gente a esquivarlo, y el rodeo es peor que el riesgo original: un grant permanente, una contraseña compartida, una copia de producción en un laptop. Un proceso que trata cada una como rutina no enseña a nadie a mirar dos veces, y la sentencia equivocada de cada cien llega a todas las filas. El camino seguro tiene que ser el fácil.

La tercera dificultad es que la unidad de riesgo no es la persona sino la sentencia, y una sentencia casi no lleva información sobre su propio tamaño. `UPDATE invoices SET status = 'sent' WHERE id = 'inv-0042'` y la misma línea con la cláusula `WHERE` cortada se diferencian en unos pocos caracteres. Nada en el texto dice “esto tocará nueve mil filas”; solo lo dicen los datos, y los datos se leen después del hecho a menos que el proceso haga antes la pregunta.

La cuarta es que la respuesta no puede ser solo una política. “Cuenta las filas antes de actualizar” se lee una vez y se cumple hasta el primer día de mucho trabajo, así que tiene que estar incorporado en la ruta: un runner que se niega a hacer commit cuando el conteo no cuadra, y una sesión que no puede escribir de ninguna otra manera.

Las consultas ad hoc tienen un problema más discreto. Una **consulta ad hoc** es una consulta que una persona escribe y ejecuta una vez, fuera de cualquier aplicación. En una sesión de solo lectura no puede cambiar nada, pero sí puede leer muchísimo, puede resultar costosa de ejecutar para la base de datos, y sus resultados salen del sistema en una captura de pantalla, una hoja de cálculo o un mensaje de chat.

Por defecto: una sesión de solo lectura

Una sesión humana empieza como un login que es miembro de `human_readonly`, conectado a través del access gateway a `ledgerly_replica_2`. Puede leer las tablas que cubre el rol, con las columnas Restricted retenidas y las columnas Confidential enmascaradas donde lo indica el Capítulo 12. No puede escribir, y nada de lo que la persona teclee cambiará eso.

La mayoría de las investigaciones caben dentro de esta sesión, y la persona no necesita pedir nada ni esperar a nadie. Yo defendería este valor por defecto solo por su costo: un diseño en el que la tarea común no necesita aprobación es un diseño en el que la aprobación de la tarea poco común sigue significando algo.

Tres hábitos mantienen a salvo la réplica compartida y sus resultados. **Acota la consulta:** el statement timeout a nivel de rol (Capítulo 5) y el row cap del gateway (Capítulo 7) detienen el caso desbocado, y seleccionar primero conteos y muestras mantiene pequeños los exports accidentales. **Selecciona lo mínimo que responda la pregunta:** `id, status, due_on` es mejor que `SELECT *`, en parte porque es más rápido y en parte porque deja las columnas Confidential fuera del resultado a menos que la pregunta las necesite. **Trata los resultados como copias de datos:** un resultado pegado en un ticket o en una hoja de cálculo es una copia de datos

de producción bajo reglas de acceso distintas (el Capítulo 13 trata esas copias como puertas laterales), así que pega identificadores y conteos, no filas con contenido personal, y nombra la consulta que los produjo.

Una salvedad importa para la sección siguiente. La réplica va por detrás del primary en una medida que suele ser pequeña y a veces no. Un conteo tomado allí es una estimación de lo que el primary contiene ahora. Para investigar, eso basta. Para decidir cuántas filas tocará una escritura, no, y el procedimiento de corrección de datos que sigue toma el conteo definitivo en el primary, dentro de la transacción que hace la escritura.

Escribir es un acto distinto

Una lectura se puede repetir, abandonar e ignorar. Una escritura cambia lo que ve todo otro principal, incluidos los servicios que se están ejecutando en ese momento y que actuarán sobre el nuevo estado antes de que nadie lo compruebe. Un `status` mal actualizado puede disparar un correo de recordatorio; un `due_on` mal actualizado puede cambiar lo que se le dice a un cliente que debe. Por eso el libro da a la escritura su propia ruta, con su propia aprobación, su propio límite de tiempo y su propio registro.

Esa ruta es el acceso just-in-time del Capítulo 8, con una propiedad que aquí importa: una solicitud de escritura está ligada a una corrección revisada. La solicitud lleva el `fix_id` de la corrección y un `approved_hash`, y sus tripletas (privilegio, tabla, columnas) se derivan de la corrección, no las teclea quien solicita. Lo máximo que puede pedir una solicitud es el techo definido por `human_jit_write`. Tres consecuencias determinan cómo se escribe la corrección.

Primera, **la solicitud se construye a partir de la corrección**. Una corrección que actualiza `invoices.status` necesita exactamente una tripleta: `UPDATE` sobre `invoices` para la columna `status`. La salvaguarda de la sección siguiente lee antes de escribir, y esa lectura usa el `SELECT` simple sobre `invoices` que `human_readonly` ya tiene, así que no hace falta ninguna tripleta de `SELECT`. La excepción es una corrección que hace un snapshot de una columna Confidential, como `billing_address` en `customers`: bajo `human_readonly` la columna está enmascarada, y un snapshot de valores enmascarados no sirve para un rollback, así que la solicitud nombra una tripleta de `SELECT` para ella y el aprobador la ve.

Segunda, **una columna Restricted nunca aparece en la solicitud**. `iban`, `account_holder_name` y `token_value` no se pueden nombrar, y `INSERT` y `DELETE` sobre `bank_accounts` y `payment_tokens` no están disponibles, porque insertar o borrar una fila establece o destruye valores Restricted. Hay otro cambio que cabe dentro del techo y que aun así no es una corrección de datos ordinaria: el `UPDATE` de `customer_id` en `bank_accounts` o `payment_tokens`. La columna no es Restricted, pero mover una fila cambia de quién es esa cuenta de pago o ese token de pago. La política de ejecución pone un gate a este cambio del vínculo con el propietario y lo encamina a la ruta de producto descrita más abajo (Capítulo 9). Si una corrección necesita cualquiera de estas cosas, no es una corrección de datos.

Tercera, **el aprobador ve la sentencia, no solo las tripletas**. El Capítulo 9 enuncia esta regla de aprobación ligada a la sentencia y cómo la hace cumplir el gateway. En una corrección de datos, la pantalla de aprobación muestra la sentencia, el filtro y el conteo esperado junto a las tripletas, y una solicitud de todas las columnas es una solicitud de otra clase de confianza. Las correcciones escritas con un alcance estrecho se aprueban más rápido, que es el incentivo que quieres.

El registro de solicitud de la corrección de las facturas atascadas, en la notación independiente del motor del Capítulo 3:

```
// request record for the stuck-invoice fix; values are illustrative
access_request:
  id          "req-0187"
  kind        "jit"
  requester   "robin"
  approver    "alex"
  fix_id      "FIX-0031"
  approved_hash "a41c...07" // set at approval, from the fix's
merge record
  reason      "reset invoices left in 'sending' by worker bug"
  expires_at  now() + 1 hour

access_request_privilege: // derived from FIX-0031, not typed
  ("req-0187", "UPDATE", "invoices", "status")
```

En los motores sin grants a nivel de columna, las columnas Restricted ya viven en tablas separadas fuera del techo (Capítulo 3). Una solicitud JIT sobre la tabla principal es entonces un grant a nivel de tabla, y el aproba-

dor ha aprobado todas las columnas no Restricted de ella, incluidas las Confidential. El grant ya no limita el alcance de la sentencia, así que la sentencia misma carga con más de la revisión.

Anatomía de una corrección de datos

Una corrección de datos es un procedimiento corto con un orden fijo. El orden importa más que cualquier paso suelto, porque cada paso existe para hacer más barato acertar con el siguiente.

1. **Enuncia la intención.** Una frase: qué estado es incorrecto, qué estado es el correcto y qué filas entran en el alcance. Si no se puede escribir en una frase, probablemente la corrección son dos correcciones.
2. **Cuenta las filas afectadas con un `SELECT`.** “Cuarenta facturas” y “nueve mil” son conversaciones distintas, y el conteo es el primer lugar donde se delata un error.
3. **Anota el número de filas esperado.** La corrección declara el número que espera cambiar. Esta es la salvaguarda.
4. **Toma un snapshot previo.** Las claves primarias y los valores antiguos de exactamente las columnas que cambiará la corrección, guardados en un lugar que sobreviva a la sesión.
5. **Ejecuta el cambio dentro de una transacción.** Compara el número de filas cambiadas con el esperado. Si difieren, haz rollback.
6. **Comprueba una poscondición dentro de la misma transacción.** Una consulta que demuestre que el estado ahora es el que dijo la intención, por ejemplo “ninguna factura de la lista conserva el estado `sending`”.
7. **Haz commit y luego registra.** El commit es el punto a partir del cual otros principals ven el cambio. El estado posterior, leído dentro de la transacción, pasa al registro junto al snapshot previo.

El pseudocódigo pone la salvaguarda en el runner y no en la memoria de la persona. Trata la base de datos, el almacén de snapshots y el registro de auditoría como sistemas separados, de modo que ninguna transacción cubre más que la base de datos. Al runner se le entrega un id de solicitud y nada más; la primera función decide si esa solicitud puede ejecutarse siquiera.

```

function loadApprovedFix(request_id, session):
    request = loadRequest(request_id)      // the gateway's record,
not the runner's input
    if request.status != "approved" or request.expires_at <= now():
        raise RequestNotLive
    if request.requester_login != session.login:    // granted to this
session's login only
        raise RequestNotForThisSession
    fix = loadFixScript(request.fix_id)
    // approved_hash was set at approval from the fix's merge record,
so
    // the script is compared with the record, not with itself
    if hashOf(fix.statements) != request.approved_hash:
        raise ScriptNotReviewed
    if request.privileges != fix.privileges:
        raise GrantsDoNotMatchFix
    return fix

```

La segunda función la ejecuta, y todo lo que hace está acotado por la primera:

```

function runDataFix(request_id, session):
    fix = loadApprovedFix(request_id, session)
    // the authoritative count, on the primary, before anything is
written
    before = readRows(fix.target_filter, fix.columns, using: session)
    if count(before) != fix.expected_rows:
        raise RowCountMismatch
    saveSnapshot(fix.id, before)
    transaction:
        // re-read under the lock: a changed value must fail, not pass
        locked = readRowsLocked(fix.target_filter, fix.columns, using:
session)
        if locked != before:
            rollback()
            raise RowsChangedSinceSnapshot
        // one source: the change runs on the same filter that was
counted
        changed = execute(fix.change, where: fix.target_filter, using:
session)
        if changed != fix.expected_rows or not
fix.postcondition(session):
            rollback()
            raise FixGuardFailed
        after = readRows(keysOf(before), fix.columns, using: session)
        // commit point: other sessions see the change only after this
line
        commit()

    try:
        saveAfterState(fix.id, after)
        auditLogAppend({ actor: session.login, action: "data-fix",
                        request: request_id, fix: fix.id, rows:
changed })
    on AuditLogUnavailable or SnapshotStoreUnavailable:
        queueForRetry(fix.id)
        notify("platform", "fix committed, record pending: " + fix.id)

```

Varios detalles de estos bloques son deliberados.

El runner nunca aporta su propio hash. Carga la corrección que nombra la solicitud y compara el hash de sus sentencias con el `approved_hash` del registro de la solicitud, que el Capítulo 9 rastrea hasta el registro del merge. Volver a teclear la sentencia cambia el texto entre la revisión y la ejecución, y la revisión quedó ligada al texto anterior.

El snapshot se guarda antes de la transacción. Un fallo del almacén de snapshots detiene entonces la corrección sin que nada haya cambiado. El costo es que las filas podrían cambiar entre el snapshot y la escritura: un

worker puede pasar una factura de `sending` a `failed` en ese intervalo, y la clave seguiría coincidiendo con el filtro.

La transacción vuelve a leer las filas bajo un bloqueo. Compara el resultado completo, con las columnas que cambia la corrección, contra el snapshot. Cualquier diferencia en el conjunto de filas o en un valor aborta la ejecución con `RowsChangedSinceSnapshot`, y quien la escribió empieza de nuevo desde un conteo fresco. En los motores que no pueden bloquear filas por un filtro, la relectura se ejecuta igualmente dentro de la transacción; reduce el intervalo y no lo cierra, y lo que queda es el conteo esperado y la poscondición.

El registro se escribe después del commit. El almacén de snapshots y el registro de auditoría son otros sistemas, así que el estado posterior se guarda una vez hecho el commit. Es el orden opuesto al del grant JIT del Capítulo 8, donde un grant no debe existir sin su entrada de log.

El orden es defendible aquí porque el gateway de Ledgerly está en la ruta de datos (más abajo): reenvía las sentencias del runner y registra la sesión, de modo que un cambio confirmado queda registrado aunque falle el append de auditoría. La entrada se reintenta y se avisa al equipo de plataforma. En un gateway que solo hace de intermediario de credenciales no habría un log de sesión de las sentencias, y la entrada de auditoría tendría que ir primero.

Un dry-run muestra lo que un conteo no puede. El conteo es un `SELECT`, así que no ve lo que la sentencia pone en marcha: un trigger que escribe en otro lugar, una cascada, una regla. Un dry-run real ejecuta el cambio dentro de una transacción y hace rollback:

```

function dryRunDataFix(request_id, session):
    fix = loadApprovedFix(request_id, session)
    if count(readRows(fix.target_filter, fix.columns, using: session))
    != fix.expected_rows:
        raise RowCountMismatch      // a wrong filter never reaches
the locks
    transaction:
        countsBefore = countsIn(fix.watch_tables, using: session)
        changed = execute(fix.change, where: fix.target_filter, using:
session)
        report = { rows: changed,
                    sample: sampleAfterValues(fix, using: session),
                    related: diff(countsBefore,
countsIn(fix.watch_tables, using: session)) }
        rollback()      // nothing is kept; the row locks are held
until here
    return report

```

El informe contiene el número de filas, una muestra de los valores posteriores y el cambio en los conteos de filas de las `watch_tables` de la cabecera de la corrección. Necesita una solicitud aprobada y los grants de escritura, así que no puede existir mientras el pull request está en revisión; se ejecuta dentro de la ventana aprobada, mantiene los mismos bloqueos de fila mientras corre y no puede ver efectos fuera de la base de datos, como un correo. Donde el costo de una sorpresa es alto, quien escribió la corrección lo ejecuta en los primeros minutos de la ventana, adjunta el informe a la solicitud, y el aprobador lo lee antes del paso de commit, al precio de una pausa.

Una salvaguarda es un mecanismo solo si la sesión no puede escribir de ninguna otra manera. El gateway de Ledgerly está en la ruta de datos (Capítulo 7) y, en una sesión de escritura JIT, reenvía escrituras únicamente desde la conexión del runner, de modo que la comprobación del hash, el conteo y el snapshot no se pueden omitir. La transacción sigue siendo del runner: el gateway reenvía cada sentencia dentro de ella y cuenta las filas a lo largo de toda la corrección, pero nunca hace commit en nombre del runner, y por eso una poscondición fallida revierte todas las sentencias. El Capítulo 9 cubre cómo comprueba el gateway cada sentencia contra el registro del merge y dónde termina su inspección. El precio es que toda escritura JIT depende de que el runner y el gateway estén disponibles. Con un gateway que solo hace de intermediario de la credencial, el runner es una convención, no una protección contra una sentencia sin cláusula `WHERE`.

Adónde va el snapshot

El snapshot previo es una copia de datos, y necesita un responsable y un vencimiento como cualquier otra. Un snapshot de valores de `status` en `invoices` son datos Internal. Un snapshot de una columna Confidential, como `billing_address` en `customers`, es Confidential, y el almacén necesita reglas de acceso al menos tan estrictas como las de la tabla de origen.

Guardaría los snapshots junto con el registro de la corrección, con un período de retención que fije el equipo de plataforma: lo bastante largo para poder responder a “¿era correcta esta corrección?”, lo bastante corto para que el snapshot no se convierta en una segunda copia, sin gobierno, de los datos de clientes. Eso son días para estados de facturas y quizá un ciclo de facturación completo para una dirección de facturación. Elige un número, anótalo y deja que venza.

El plan de rollback

Un snapshot previo no es un plan de rollback hasta que alguien ha escrito cómo se usaría. El plan son una o dos frases en la cabecera de la corrección, como “restaurar `status` desde el snapshot para las mismas claves”, y debería ser posible convertir el snapshot en un script de corrección de forma mecánica.

El rollback es en sí mismo una corrección de datos. Pasa por la misma solicitud y la misma revisión, porque otros principals pueden haber actuado sobre el nuevo estado entretanto y restaurar los valores antiguos puede sobreescribir actualizaciones legítimas. Por eso el runner de un rollback comprueba que los valores actuales siguen coincidiendo con el estado posterior registrado por la corrección original, y se niega a tocar las filas que han cambiado desde entonces. Una corrección inversa que no puede decir “estas filas han seguido su camino, trátalas por separado” provoca la siguiente reparación.

Algunos cambios no tienen rollback, y la cabecera debe decirlo. Un correo, un webhook o una llamada a un proveedor de pagos no se pueden deshacer restaurando una columna. Para esos casos, la conversación ocurre antes de que se ejecute la corrección: ¿qué efectos dispara este cambio de estado, y deberían pausarse antes los servicios que reaccionan?

Dos tablas no tienen rollback en el sentido ordinario, porque ningún rol humano puede editarlas. `ledger_entries` y `audit_events` son append-only. Una corrección del libro mayor es una entrada nueva, emitida solo mediante la ruta de corrección de `payments-service`, nunca por un login humano, de modo que una corrección que parece necesitar un cambio ahí es una solicitud de esa ruta, no de un grant. Un trigger que rechaza todo `UPDATE` y `DELETE` sobre ambas tablas hace que la regla se mantenga incluso si un grant se amplía por error. Tiene un costo: `ledgerly_owner` todavía puede eliminar el trigger, de modo que el verdadero punto de revisión sigue estando en las migraciones que tocan estas tablas (Capítulo 11).

Un DELETE es el mismo procedimiento con un rollback más difícil

La anatomía anterior está escrita para `UPDATE`. Supón que un retry en `invoice-worker` insertó filas duplicadas en `invoice_lines` y la reparación las elimina. `DELETE` sobre `invoice_lines` está dentro del techo de `human_jit_write`, y se diferencia en tres aspectos.

El grant acota la tabla y no las filas, porque una tripleta `DELETE` no tiene columnas, así que el límite viene del filtro, del conteo esperado y de la revisión, y el filtro debería ser una lista de claves. El snapshot es la fila completa, ya que nada más guardará los valores después, y una tabla con columnas Confidential hace que el snapshot sea tan sensible como la tabla. El rollback es un re-insert, que es otra corrección de datos con su propia solicitud. Una tripleta `INSERT` nombra sus columnas, así que la solicitud del re-insert decide si `id` y `created_at` están entre ellas. Si no lo están, la fila restaurada recibe valores nuevos para ambas, y nada de lo que hacía referencia a los identificadores antiguos se restaura. Si lo están, el aprobador ve que se escribe una clave y una marca de tiempo, algo que no hace ninguna corrección ordinaria, y lo que el motor haga con una clave suministrada depende de cómo genere las claves. La cabecera indica qué opción toma el rollback.

```

fix:
  id          "FIX-0044"
  intent      "Remove the duplicate invoice_lines rows created
by a retry.
              No other row is deleted."
  target_filter "id IN (<ids of the duplicates>)"
  change       "DELETE"
  expected_rows 12                      // illustrative
  snapshot     "whole rows"
  privileges    ("DELETE", "invoice_lines") // no columns for
DELETE
  rollback     "re-insert from the snapshot as a new fix; its
INSERT triple
              leaves out id and created_at, so the rows get new
values and
              references to the old ids are lost"
  effects      "invoice totals are recomputed by the worker on
next run"
  watch_tables ("invoices")

```

Donde perder los identificadores no es aceptable, quien revisa pide una protección más barata que el rollback: una clave foránea que rechace el borrado mientras otra fila siga apuntando a la línea, o una reparación distinta que no elimine la fila.

Revisión: four-eyes, en un repositorio

Una corrección de datos es código que se ejecuta una vez, así que recibe el control en el que los ingenieros ya confían: un pull request. La corrección es un script en un repositorio, con una cabecera, un revisor y un historial, y el historial responde, seis meses después, quién cambió un estado y por qué.

```
// data-fixes/2026-10-06-FIX-0031-reset-sending-invoices.fix
fix:
  id          "FIX-0031"
  intent      "Invoices stuck in 'sending' after the worker bug
              are set to 'sent'. No other column changes."
  target_filter "status = 'sending' AND updated_at < '<cutoff>'"
  change       "SET status = 'sent'" // applied to
target_filter only
  expected_rows 40
  postcondition "no invoice matches target_filter"
  privileges     ("UPDATE", "invoices", "status")
  rollback       "restore status from snapshot for the same ids,
                  only where status is still 'sent'"
  effects        "reminders resume for these invoices"
  watch_tables   ("invoices") // read by the dry run
  author         "robin"
  reviewer      "alex"
```

El campo `privileges` conecta el repositorio con el Capítulo 8. Las tripletas de la solicitud se copian de él, de modo que el aprobador mira la solicitud y la corrección lado a lado, y `loadApprovedFix` rechaza una solicitud cuyas tripletas difieren de las de la corrección. Una sentencia que toca una columna que sus privilegios no nombran falla en tiempo de ejecución, que es el resultado buscado; el `grant` es la segunda línea de defensa detrás de la revisión.

Qué lee el revisor

El trabajo del revisor es más estrecho que “¿es esto una buena idea?”.

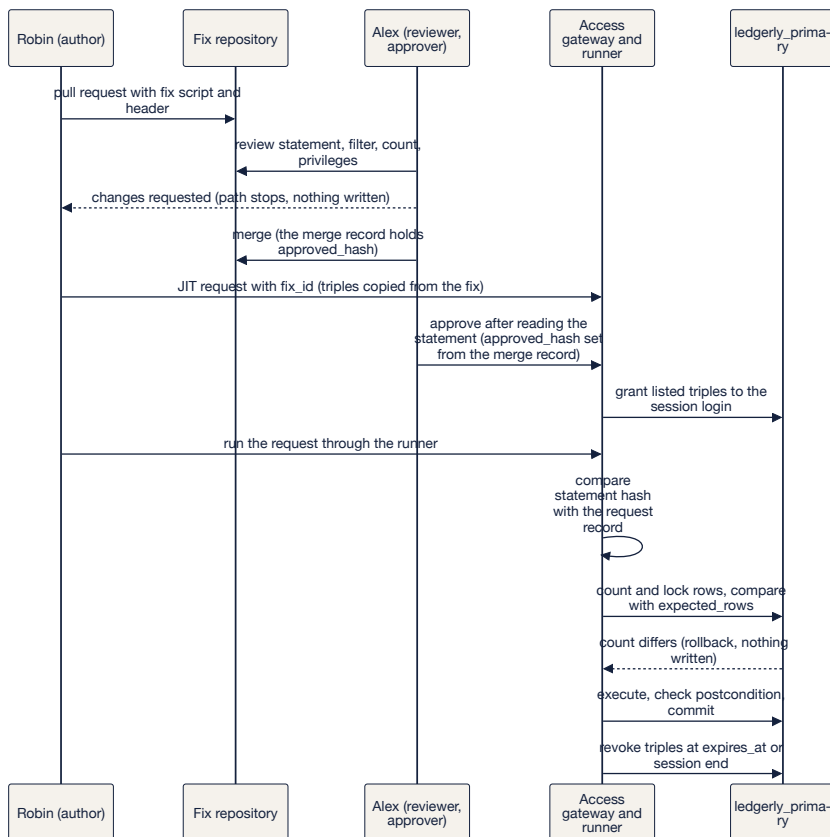
- **¿La sentencia coincide con la intención?** Deben coincidir en las tablas, las columnas y el filtro.
- **¿Es específico el filtro?** Un filtro que selecciona por clave es el más seguro; un filtro que selecciona por estado y un límite de tiempo es aceptable si el conteo cuadra; un filtro sin ninguna restricción se rechaza a primera vista.
- **¿Es plausible el conteo esperado, y de dónde salió?** Un revisor que ve `expected_rows: 40` puede pedir la salida de la consulta de conteo en el pull request. Un número que nadie puede rastrear hasta su origen es un número que nadie comprobó.
- **¿Los privilegios nombran solo lo que necesita la sentencia?** Compáralos línea por línea.

- **¿La corrección toca una columna Restricted, una tabla append-only o una tabla fuera del techo?** Si es así, no es una corrección de datos, y la revisión termina con una indicación hacia la ruta correcta.
- **¿Qué dispara este cambio?** Recordatorios, correos, jobs que leen la columna. Esta pregunta atrapa los casos discretos.
- **¿La vista previa concuerda con la intención?** El pull request contiene la salida de la consulta de conteo y una vista previa con `SELECT` de las filas y de los valores posteriores; el informe del dry-run se adjunta después a la solicitud, y un trigger o una cascada aparece ahí y en ningún otro sitio.
- **¿Es utilizable la frase del rollback?** Para un `DELETE`, dice qué no puede restaurar el re-insert.

La revisión es four-eyes en el sentido del libro: una segunda persona revisa antes de que se ejecute una acción riesgosa. El merge fija el texto: un cambio en el script después de la revisión produce un hash distinto, y el runner lo rechaza. El aprobador de la solicitud JIT y el revisor de la corrección son dos actos. En un equipo grande preferiría dos personas, para que el aprobador compruebe el acceso contra la corrección en lugar de releerla; en un equipo pequeño serán a menudo la misma persona, lo cual es aceptable si ninguna de las dos es quien la escribió. Quien escribe nunca aprueba su propia corrección ni su propio grant.

La secuencia

El diagrama muestra la ruta desde una solicitud hasta un cambio confirmado. Busca los lugares donde la ruta puede detenerse sin que se haya escrito nada: la revisión, la comparación de hashes y las salvaguardas del runner.



Cuando el cambio es en una columna Restricted

El techo prohíbe las columnas Restricted en cualquier solicitud JIT (el Capítulo 3 da la razón), así que toda una clase de reparaciones no tiene ruta SQL. Imagina que el `account_holder_name` de un cliente en `bank_accounts` se introdujo con una errata. Una persona no puede corregirlo en una sesión JIT, porque la columna no puede nombrarse en una solicitud. Tampoco puede hacerlo una sesión break-glass: `human_breakglass` no tiene privilegio de escritura sobre columnas Restricted, diga lo que diga el proceso.

La ruta es la aplicación. `admin-console` le pide a `payments-service`, el único que escribe en `bank_accounts`, que haga el cambio; `admin-console` exige que una segunda persona lo apruebe antes de que `payments-service` lo realice, y el cambio queda registrado en `audit_events`. Es una

ruta de producto, no una corrección de datos: tiene un formulario, una validación, un campo de motivo, un aprobador y un registro, y se ejecuta con el rol propio del servicio, no con el de una persona.

Esto cuesta más que una sentencia SQL, y creo que debe ser así. Esos valores son aquellos cuya exposición daña a los clientes de maneras difíciles de deshacer, y la aplicación es el único lugar donde viven la validación y el conjunto completo de efectos secundarios. El costo son minutos para quien necesita el cambio; el ahorro es que una reparación tecleada a mano ya no puede introducir un número de cuenta bancaria mal formado. Un token incorrecto en `payment_tokens` también lo reemplaza `payments-service` mediante el flujo propio del proveedor, no se parchea.

Break-glass no es una ruta para correcciones de datos

`human_breakglass` escribe donde no llega el techo JIT, y es el único rol humano que lee columnas Restricted (sus privilegios exactos están en el Capítulo 3). Sobre el papel parece una forma cómoda de hacer una reparación que la ruta JIT rechaza. No lo es, y la razón es lo que le falta al rol.

Break-glass es autodeclarado. No tiene aprobador, ni `fix_id`, ni runner, y sus salvaguardas son el conjunto de logs y la revisión obligatoria que define el Capítulo 8. Una acción limitadora que detiene un daño en curso, revisada después, es para lo que sirve el rol (el Capítulo 15 cubre el lado de los incidentes). La reparación duradera de los datos es un acto distinto y es siempre una corrección de datos: ejecutada en una sesión break-glass, se salta el único control que lee la sentencia antes de que se ejecute, en un rol donde es posible un `UPDATE` sin salvaguarda de filas. El Capítulo 8 enuncia eso como un riesgo aceptado, y encaminar las reparaciones por el rol lo volvería rutina.

La segunda razón son los incentivos. Si break-glass puede hacer una reparación, se convierte en el lane rápido, y su alerta deja de significar algo. Una corrección de datos que “necesita” break-glass señala otra cosa: el ciclo de aprobación nocturno es demasiado lento, al techo le falta una columna que la corrección necesita legítimamente, o la corrección pertenece a la ruta de la aplicación. Si una reparación no puede esperar, detén el daño con la acción disponible más pequeña y ejecuta la corrección de datos por la ruta normal cuando pase la presión.

Convertir una corrección repetida en una funcionalidad del producto

Una corrección de datos es un síntoma: es alcanzable un estado que la aplicación no puede deshacer. La primera vez es una corrección; la tercera es una solicitud de funcionalidad que nadie ha registrado. Mantendría un directorio de scripts de corrección, etiquetados por el tipo de reparación, y cuando la misma etiqueta aparezca de nuevo, alguien pregunta si pertenece a `admin-console`. No hay nada preciso en “tres veces”: una reparación de diez filas dos veces al año quizá nunca justifique una pantalla, mientras que una que se ejecuta cada semana, necesita una revisión cada vez y podría hacerla soporte en un minuto es una funcionalidad esperando a que la escriban.

Toma las facturas atascadas del comienzo, y supón que el estado se repite. La tabla de grants del Capítulo 3 ya muestra lo que `app_admin_console` puede hacer sobre `invoices`: `SELECT` y `UPDATE` solo sobre `status`. La versión de producto es una acción en la página de la factura, disponible para un grupo reducido, que restablece el estado de una factura, pide un motivo y escribe la acción en `audit_events` con la identidad del principal humano. Lleva sus propias comprobaciones: solo facturas en el estado atascado, solo una a la vez, y un límite de cuántas puede restablecer una persona por hora.

La conversión traslada el riesgo desde la sentencia, que paga una solicitud, una revisión y un runner en cada reparación, hacia el código, que se revisa, se prueba y se versiona una sola vez. Soporte puede usar la acción sin una sesión de base de datos, de modo que cada reparación que se convierte en funcionalidad es una escritura recurrente que ya no necesita acceso humano.


```

function resetStuckInvoice(actor, invoice_id, reason):
  if not actor.canResetInvoices:
    raise NotAuthorized
  if reason is empty:
    raise ReasonRequired
  if countResetsBy(actor, within: "1 hour") >= resetLimit:
    raise RateLimited

  transaction:
    invoice = readInvoiceLocked(invoice_id)
    if invoice.status != "sending" or invoice.updated_at >
cutoffForStuck():
      rollback()
      raise NotStuck
    setInvoiceStatus(invoice_id, "sent")
    // commit point: the status change and the business record
agree
    appendAuditEvent({ actor_principal: actor.id, action: "reset-
stuck-invoice",
                      subject_id: invoice_id, reason: reason })
    commit()

```

El evento de auditoría y el cambio de estado están ambos en `ledgerly_primary`, así que una sola transacción los cubre; `audit_events` es el registro de negocio de la aplicación, no el registro de auditoría del Capítulo 14. En el runner de más arriba, el registro de auditoría es un sistema aparte y el orden de las operaciones es lo que aporta la seguridad.

No toda reparación debería ser una funcionalidad. Un botón que cualquiera puede pulsar tiene un perfil de riesgo distinto al de un script que revisaron dos personas, así que las reparaciones que cambian dinero, tokens o datos bancarios se quedan en una ruta four-eyes aunque sean frecuentes. Una reparación se convierte en una acción de autoservicio solo si su peor caso está acotado por el diseño de la acción; de lo contrario se convierte en una solicitud con una aprobación dentro del producto. La prueba es si la reparación se repetirá y si una persona es hoy la única que puede hacerla. Si ambas respuestas son afirmativas, el sprint es una inversión y no un costo.

Runbooks para las reparaciones que puedes predecir

Algunas correcciones se repiten en su naturaleza aunque cambien los detalles: reasignar una factura al cliente correcto, reproducir un export fallido, reparar una fila duplicada. Para estas, un **runbook** es el artefacto adecuado: un procedimiento escrito y seguido para una tarea repetible. Tiene el mismo esqueleto que un script de corrección, con los parámetros en blanco. Nombra el síntoma que lo activa, las consultas de diagnóstico que lo confirman, la plantilla de la corrección y los privilegios que necesita, el origen del conteo esperado, los efectos que hay que comprobar, el rollback, y quién puede escribirlo y revisarlo. También dice cuándo parar y escalar: “si el conteo del diagnóstico supera el número de este párrafo, este no es el problema conocido”.

Un runbook también puede llevar una preaprobación, para que una reparación predecible no espere a un aprobador de noche. Cinco condiciones lo deciden, y las comprueba el runner, no la persona:

- el filtro selecciona por clave, una lista de ids, y no por un estado y un límite;
- el conteo de filas esperado es como máximo N, un número que el runbook fija para su propia reparación (digamos, 20 para facturas atascadas; ilustrativo);
- las tripletas son exactamente el conjunto fijo del runbook;
- la reparación no dispara ningún efecto saliente, como registra el campo **effects** del runbook;
- la sentencia es la plantilla del runbook con solo la lista de claves sustituida, y el hash de la plantilla se aprobó cuando se revisó el runbook.

El registro del merge del runbook almacena ese hash de la plantilla. La lista de claves es la única parte que lleva la solicitud, y el runner la sustituye en la plantilla almacenada y comprueba el resultado contra las tripletas de la solicitud; nunca toma un hash ni una sentencia de quien lo invoca (el Capítulo 9 cubre cómo comprueba el gateway la sentencia sustituida). Si falla cualquier condición, la solicitud vuelve a la revisión completa con un aprobador.

El costo es que una ejecución preaprobada se realiza sin que nadie lea esa noche la lista de claves, de modo que el límite de filas del runbook acota una lista equivocada, y alguien distinto del runner lee el registro el siguiente día laborable. Un runbook también envejece: uno para una tabla que ha cambiado de forma da a una sentencia obsoleta la apariencia de haber sido revisada. Liga cada runbook a una fecha y a un responsable, y trata una ejecución fallida como una indicación para revisarlo, no como una molestia que esquivar.

Un UPDATE sin WHERE, analizado sin culpas

Considera Ledgerly en su estado inicial. Una persona se conecta con un cliente de escritorio que recuerda la contraseña de administrador compartida y hace auto-commit. La tarea son las facturas atascadas. La sentencia está escrita en dos líneas en el editor: el `UPDATE` en la primera, el `WHERE` en la segunda. El cliente ejecuta la selección resaltada, y la selección cubre solo la primera línea. La sentencia se ejecuta, y todas las filas de `invoices`, digamos nueve mil, quedan con el estado `sent`.

Una versión de esta historia termina con alguien a quien se le dice que tenga más cuidado. No enseña nada, porque la siguiente persona sentada en la misma silla se enfrenta a la misma distancia entre una pregunta y un cambio irreversible. La pregunta útil es qué en el diseño permitió que una sola sentencia llegara a todas las filas.

- **El login podía escribir en todas partes.** El rol de administrador compartido es propietario de todas las tablas, así que un error no quedaba confinado a la tabla ni a la columna que necesitaba la tarea.
- **El cliente hacía commit por su cuenta.** Nadie vio “nueve mil filas afectadas” antes de que la sentencia se volviera definitiva.
- **Nunca se pidió el conteo.** Nada exigía que la persona supiera cuántas filas esperar, así que el número impreso después del hecho no significaba nada.
- **Nadie más había leído la sentencia.** El único lector era quien la escribió, que leyó la intención dentro del texto.
- **No había snapshot previo.** Los estados antiguos se habían perdido, y la reparación de la reparación no tenía punto de partida.

- **El log decía “el rol compartido”.** La pregunta “¿quién ejecutó esto?” solo tenía respuesta a través del historial del chat.

La recuperación parte de un backup, concilia lo que haya ocurrido desde entonces e informa a todos los que están aguas abajo (el Capítulo 13 y el Capítulo 15 tratan esa mecánica). El diseño fijó su costo antes de que se tecleara la sentencia.

Ahora recorre la misma tarde en el estado objetivo. Robin escribe la corrección con su cabecera y un conteo esperado de 40, y Alex la revisa. La solicitud JIT nombra `UPDATE` sobre `invoices`, columna `status`, y el grant dura una hora. Si se hubiera quitado el filtro del script revisado, el runner contaría nueve mil frente a los 40 esperados y se negaría antes de ejecutar ninguna sentencia. Una sentencia tecleada en el cliente propio de la persona no se reenviaría como escritura (la regla del gateway del Capítulo 9), y en una sesión simple de `human_readonly` el `UPDATE` no se ejecutaría en absoluto.

La salvaguarda no hace correcta una corrección. Hace ruidosa una clase de error: la sentencia tocó un número de filas distinto del que dijo quien la escribió. Una sentencia que toca exactamente 40 filas, pero las 40 equivocadas, pasa, y por eso la revisión, la poscondición y el snapshot existen como capas separadas, y por eso el Capítulo 9 dice lo que el análisis no puede ver. La salvaguarda también vale lo que vale su número. Un conteo esperado copiado de la propia salida de la ejecución enseña al runner a darse la razón a sí mismo, así que el número tiene que salir de una consulta que quien revisa pueda ejecutar de forma independiente, idealmente antes de haber escrito la sentencia. Donde el motor tiene grants a nivel de columna (Capítulo 3), un grant estrecho limita hasta dónde pueden llegar los casos que la salvaguarda no detecta.

Trade-offs y costos

El procedimiento es más lento que teclear una sentencia. Una solicitud, una revisión, una aprobación y un runner añaden minutos en un buen día y mucho más cuando el aprobador está durmiendo. Un jueves por la tarde eso es aceptable. De noche, bajo una alerta, es el costo que tienta hacia break-glass, y merece atención de diseño: un revisor en la rotación,

la preaprobación por runbook de más arriba para las reparaciones que la merecen, y un límite declarado de lo máximo que el equipo esperará una aprobación nocturna, escrito antes de la primera noche en que haga falta.

La revisión puede volverse un reflejo. Una aprobación dada en segundos añade fricción sin reducir la exposición; el Capítulo 8 desarrolla el argumento completo. La señal específica de este capítulo es el pull request: si quienes revisan nunca piden la salida de la consulta de conteo y nunca solicitan un cambio en una sentencia, o no están leyendo nada o solo están revisando correcciones perfectas. Las defensas son los puntos de la lista de verificación que exigen un artefacto: la salida del conteo y la vista previa en el pull request, el informe del dry-run en la solicitud y una comparación explícita de la sentencia con los privilegios.

Las transacciones mantienen bloqueos. El runner inicia la transacción después de la revisión y la termina en segundos. Un procedimiento que la abre primero, “por seguridad”, y deja que la persona decida después hace esperar a otros escritores, llena las conexiones del pooler y hace que los timeouts del Capítulo 5 salten sobre trabajo que no tiene relación. Mantén la decisión humana fuera de la transacción.

Las correcciones grandes necesitan lotes. Una corrección que toca millones de filas no puede ejecutarse como una sola transacción sin bloqueos prolongados y un registro de transacciones grande. La respuesta habitual es dividirla en lotes, digamos unos pocos miles de filas cada uno (un tamaño ilustrativo, no una recomendación), cada uno con su propio conteo y su propia salvaguarda, con un modo por lotes del runner que repite las mismas comprobaciones. Un fallo a mitad de camino deja los datos en un estado mixto, así que la corrección tiene que escribirse para poder volver a ejecutarse: el filtro selecciona solo las filas que siguen en el estado antiguo. Una reparación grande sigue siendo una corrección de datos; solo la primera población de una estructura nueva pertenece a una migración (Capítulo 11).

Los snapshots y el repositorio de correcciones necesitan un responsable. Un snapshot sin vencimiento ha cambiado un riesgo por otro, y para las columnas Confidential el segundo puede ser el mayor. Un directorio de scripts de corrección es un registro de cómo derivó producción, y sin un índice por etiqueta nadie buscará la ocurrencia anterior de un problema.

Cuándo no usar todo esto. En un equipo pequeño con una base de datos que no contiene nada sensible, un pull request por cada cambio de estado es más ceremonia de la que justifica el riesgo; contar primero y usar una transacción siguen compensando. En una base de datos de desarrollo o de staging sin datos de producción, nada de esto se aplica, y para una copia que se reconstruye cada noche, como `ledgerly_analytics`, una escritura equivocada cuesta una reconstrucción. El procedimiento escala con el costo de equivocarse, y el primary es donde ese costo es más alto.

Errores comunes

- **Tratar la cláusula `WHERE` como la salvaguarda.** Es tentador porque la cláusula es donde vive la intención de quien escribe. Pero una cláusula es texto, y el texto se puede cortar, teclear mal o ejecutar en parte. El remedio es una salvaguarda que compare el conteo de filas cambiadas con una expectativa declarada y haga rollback si no coinciden.
- **Contar en la réplica y actualizar en el primary.** La réplica se retrasa, así que los dos números pueden diferir por una buena razón. El remedio es tomar el conteo definitivo en el primary, dentro de la transacción, y usar el conteo de la réplica solo para explorar.
- **Volver a teclear la sentencia revisada.** Quien revisó aprobó un archivo, y una persona lo copió en un editor y cambió una cosa “ya que estaba”. El remedio es un runner que compare el hash del script con el `approved_hash` del registro de la solicitud, junto con un gateway que reenvíe las escrituras de la sesión solo desde ese runner. En un gateway que solo hace de intermediario de credenciales la salvaguarda es una convención, y la revisión tiene que cargar con ella.
- **Pedir un grant amplio “por seguridad”.** Convierte la aprobación en un juicio sobre la confianza en lugar de una comprobación contra una sentencia, y amplía lo que puede alcanzar un error. El remedio es pedir las tripletas que necesita la sentencia y hacer una segunda solicitud si la situación cambia.
- **Usar break-glass porque el aprobador es lento.** La ruta es más corta y la corrección es urgente. El remedio es resolver la lentitud de la aprobación nocturna y reservar break-glass para las emergencias para las que se definió.

- **Escribir una columna Restricted mediante una sesión “especial”.** Actualizar un `iban` directamente parece un favor al cliente, y elimina la validación, la aprobación y el registro de la aplicación de un solo golpe. El remedio es la ruta de la aplicación, aunque sea más lenta.
- **Guardar un snapshot que nadie gobierna.** El snapshot protege contra una mala corrección y se convierte en una copia sin gobierno de datos Confidential en una carpeta compartida. El remedio es un responsable, una regla de acceso equivalente a la de la tabla de origen y una fecha de vencimiento.
- **Omitir la frase del rollback porque la corrección es pequeña.** Las correcciones pequeñas son las que se revierten a toda prisa. El remedio es un plan de una línea, escrito antes de la ejecución, que diga qué restaura el snapshot y dónde no debe hacerlo.
- **Dejar una corrección repetida como script para siempre.** El resultado es una escritura recurrente que necesita a una persona en el primary cada vez. El remedio es mirar periódicamente las etiquetas del directorio de correcciones y tomar una decisión, aunque la decisión sea “todavía no”.
- **Revisar la corrección de la intención en lugar de la corrección de la sentencia.** Un revisor que debate si vale la pena hacer la reparación puede no leer nunca la sentencia. El remedio es una lista de verificación que haga las preguntas estrechas en orden.

Ideas clave

- Una sesión de solo lectura es la opción por defecto adecuada para investigar, y funciona solo si la tarea común no necesita aprobación.
- Escribir es un acto distinto de leer, así que pasa por su propia solicitud, revisión, límite de tiempo y registro. La solicitud lleva un `fix_id` y un `approved_hash` tomado del registro del merge, sus tripletas se derivan de la corrección y el aprobador ve la sentencia.
- El orden seguro de una corrección de datos es conteo, snapshot, transacción, salvaguarda del conteo de filas, poscondición, commit, y debe hacerlo cumplir el runner, no la memoria de la persona, con el gateway reenviando las escrituras de la sesión solo desde el runner.

- Una salvaguarda hace ruidosa una clase de error y no hace correcta una corrección, y por eso la revisión, los grants estrechos y un snapshot previo existen como capas separadas.
- Un cambio en una columna Restricted no es una corrección de datos: pasa por `admin-console` hasta `payments-service` con un segundo aprobador y un registro en `audit_events`, y break-glass no es una forma de rodearlo.
- Una corrección manual repetida es una funcionalidad de producto que falta, y cada una que se convierte en una acción de `admin-console` es una escritura recurrente que ya no necesita a una persona en el primary.
- Cuando una sentencia llega a todas las filas, la pregunta útil es qué en el diseño lo permitió, y la respuesta es una lista de salvaguardas ausentes, no una persona.

Las correcciones de datos cambian lo que hay en las tablas; el siguiente tipo de cambio altera las tablas mismas. El Capítulo 11 se ocupa de los cambios de esquema, de quién los ejecuta y con qué identidad, y del hábito de hacerlos compatibles hacia atrás para que la reparación de un día no sea la caída del siguiente.

Capítulo 11 — Cambios de esquema y acceso para migraciones

Imagina un jueves por la tarde en Ledgerly. El equipo de facturación quiere una columna nueva en `invoices` para mostrar el impuesto por separado del total, y el cambio es pequeño: una columna, un valor por defecto, un backfill para las filas antiguas. Robin, que está de on-call esta semana, revisa el pull request y lo aprueba. El pipeline ejecuta la migración como `ledgerly_migrator`, el deploy termina, y durante unos noventa segundos toda solicitud a `billing-api` que toca una factura se queda colgada y luego falla.

Nada en la migración estaba mal, y nadie hizo nada fuera de lo común. Una transacción larga de un batch de `invoice-worker`, dejada abierta mientras esperaba a un servicio externo de correo, mantenía un bloqueo sobre la tabla, la migración quedó en cola detrás de ella, y cada sentencia que llegó después de la migración quedó en cola detrás de la migración. Alex, del equipo de plataforma, encuentra la causa enseguida. Entonces llega la pregunta más difícil: la migración se ejecutó con el poder del propietario de todas las tablas, se ejecutó sin supervisión, y lo único que había entre ese poder y el libro mayor era la revisión de un diff. ¿Cómo deben llegar los cambios de esquema al primary, con qué identidad, y cómo evitamos que se conviertan en un agujero en el modelo de acceso o en una caída?

Este capítulo añade un paso del estado objetivo: las migraciones se ejecutan como `ledgerly_migrator` desde el pipeline, y nada más tiene el poder de cambiar el esquema.

En el modelo de cinco partes del Capítulo 1, una migración es **Ejecución** con el límite más amplio que existe. El pipeline tiene los privilegios del propietario por diseño, así que el riesgo no está en quién tiene una credencial. Está en lo que la migración está autorizada a hacer, en cómo se ejecu-

tan sus sentencias y en qué registro sobrevive. Para las migraciones eso significa una lista corta de clases de sentencia permitidas en cada lane (autorización), una ruta que decide cuándo y cómo se ejecuta una sentencia, con una espera de bloqueo corta y un retry (ejecución), y un registro que el propietario no puede reescribir (rendición de cuentas).

El problema y por qué es difícil

Un cambio de esquema es lo más privilegiado que hace la mayoría de los equipos a una base de datos en un día rutinario. La propiedad permite al propietario eliminar una tabla, cambiar el tipo de una columna, reescribir grants y, en muchos motores, saltarse las restricciones que se aplican a todos los demás. El Capítulo 3 dio ese poder a `ledgerly_owner`, un rol sin login, y dejó que `ledgerly_migrator` lo asumiera solo mientras se ejecuta una migración. Ese arreglo responde quién es el propietario del esquema. No responde quién puede iniciar una migración, qué tiene permitido hacer una migración en ejecución, ni qué les pasa a todos los demás mientras corre.

Tres riesgos distintos están dentro de esta única actividad.

El primero es la **autoridad**. Quien pueda activar el paso de migración del pipeline puede, en la práctica, ejecutar sentencias de nivel propietario en producción. Lo permanente es la autoridad, en el sentido que describió el Capítulo 3: una autoridad permanente de propietario, ejercida mediante una credencial de vida corta. Los límites son de procedimiento: la credencial vive solo en el pipeline, vence con el job, y el pipeline es la única forma autorizada de usarla. Esos límites valen lo que valgan los controles de acceso del propio pipeline, lo que hace del pipeline parte del límite de seguridad de la base de datos, lo haya dibujado alguien así o no.

El segundo es la **disponibilidad**. Muchos cambios de esquema necesitan bloqueos, y el bloqueo es donde ocurre el daño. Una sentencia que debe esperar un bloqueo no solo espera: en muchos motores, las sentencias que llegan detrás de ella también quedan en cola, de modo que una migración que no consigue su bloqueo en el primer segundo puede cerrar la tabla para todos durante todo el tiempo que siga abierta la transacción más an-

tigua que tiene delante. El tamaño del cambio importa poco. Un cambio de una columna puede ser tan disruptivo como una reescritura si la transacción equivocada se interpone.

El tercero es el **drift**. Un esquema que se define mediante migraciones y que además se edita a mano es un esquema que nadie puede describir por completo. Un índice de hotfix añadido desde una consola en una mala noche es útil, y un año después es una diferencia entre entornos que nadie recuerda, junto con grants que no coinciden con ningún archivo.

Cada uno se puede reducir con diseño. Un rol de migración acotado por el pipeline, una espera de bloqueo que falla rápido, una forma de cambio que nunca necesita un bloqueo largo y una comprobación que compara el esquema vivo con el esperado son cosas que se pueden construir. Una política que dice “ten cuidado con las migraciones” no está en esa lista.

Quién ejecuta DDL, y con qué rol

DDL (data definition language) es la familia de sentencias que crean, alteran y eliminan objetos del esquema, en contraposición a las sentencias que leen y escriben filas. La regla de este capítulo es breve: el DDL se ejecuta como `ledgerly_migrator`, desde el pipeline, y ningún rol de aplicación puede ejecutarlo.

Los roles de aplicación no tienen DDL

Ninguno de los roles `app_*` del Capítulo 3 es propietario de una tabla ni puede crear, alterar o eliminar una. Es fácil decirlo y algo costoso mantenerlo, porque muchos frameworks pueden crear o cambiar tablas al arrancar, y algunos equipos lo activan porque quita un paso del deploy. El costo de esa comodidad es que el servicio necesita entonces la propiedad, y un servicio comprometido o confundido puede eliminar y volver a otorgar. Desactivaría la gestión automática del esquema en producción, y la conservaría para las bases de datos de desarrollo, donde nadie tiene que explicar una tabla eliminada.

El efecto secundario visible es que un deploy que incluye un cambio de esquema son ahora dos pasos: la migración, y luego el rollout del servicio. El orden entre ambos es una cuestión de diseño que responde el patrón `expand and contract` de más abajo.

El migrator y el propietario

`ledgerly_migrator` tiene un login que usa el pipeline; es miembro de `ledgerly_owner` y no tiene privilegios de datos propios. La pertenencia da al migrator los privilegios del propietario, pero no lo convierte en el propietario. En muchos motores un objeto nuevo pertenece al rol que esté activo cuando se crea, así que una migración que se ejecutara bajo el login propio del migrator dejaría sus tablas a nombre del migrator. Por eso la regla es la misma en todos los motores: una migración siempre cambia explícitamente a `ledgerly_owner`, porque la propiedad sigue al rol activo y no a los privilegios heredados. Cada objeto que crea queda entonces a nombre de `ledgerly_owner`, y la migración vuelve al rol propio del migrator cuando termina. En una notación que no pertenece a ningún motor:

```
function runMigration(migration):
    login = fetchMigratorLogin(identity: pipeline.workloadIdentity)
    connection = connect(login, database: "ledgerly_primary")
    // lock_wait 3s, statement timeout none (Chapter 5)
    applyRoleSettings(connection, role: "ledgerly_migrator")
    if isApplied(connection, migration.id):
        close(connection)
        return // a retry never re-runs
    applied work
    // external record first: the owner can rewrite the database's own
    record
    auditLogAppend({ actor: login.name, action: "migration-start",
                     migration: migration.id, commit: migration.commit
    })

    try:
        assumeRole(connection, "ledgerly_owner")
        execute(connection, migration.statements)
        recordApplied(connection, migration.id)
    on LockWaitExceeded or DatabaseError as error:
        // owner rights end with the connection on every path
        resetRole(connection)
        close(connection)
        auditLogAppend({ actor: login.name, action: "migration-
failed",
                        migration: migration.id, error: error.kind })
        raise error
    resetRole(connection)
    close(connection)
    auditLogAppend({ actor: login.name, action: "migration-applied",
                     migration: migration.id })
```

Importan cuatro detalles. La credencial se emite para este job mediante la workload identity del pipeline en tiempo de ejecución (el patrón del Capítulo 4) y vence con el job, de modo que una variable de pipeline copiada no puede conservar durante meses una credencial de propietario que funcione y nada del repositorio guarda una. El cambio al rol de propietario se limita a las sentencias de una migración. El registro de que una migración se aplicó se escribe en la misma ejecución, de modo que la base de datos conoce su propia versión, de la cual depende la comprobación de drift de más abajo. Y la entrada externa se escribe antes de la primera sentencia de propietario, porque `recordApplied` vive en una base de datos que el propietario puede reescribir. El costo es que un almacén de auditoría no disponible detiene la migración antes de empezar. Es el orden opuesto al de la corrección de datos del Capítulo 10, donde el log de sesión del gateway ya registra las sentencias: aquí nada salvo el pipeline ve lo que hace la sesión del propietario, así que el registro va primero. Cada entrada de inicio tiene un final correspondiente, `migration-failed` o `migration-applied`, e `isApplied` hace segura una repetición incluso cuando el paso que falló es el último append de auditoría.

En algunos motores, registrar la versión y ejecutar el DDL pueden compartir una transacción, y en otros una sentencia DDL termina la transacción que la rodea. Sabe de cuál tienes. Si un fallo puede dejar el esquema cambiado y la versión sin registrar, la migración debe escribirse de modo que ejecutarla por segunda vez sea inocuo.

Las personas no tienen el migrator

Las personas nunca se conectan como `ledgerly_migrator`. Si una persona necesita un cambio de esquema, escribe una migración y el pipeline la ejecuta. Eso suena a fricción, y lo es, pero compra una propiedad útil: cada uso del poder del propietario corresponde a una ejecución del pipeline, que tiene un commit, una revisión y un log. Cuando la credencial del migrator también está en el gestor de contraseñas del equipo, esa propiedad desaparece, y con ella la razón de tener el migrator.

Qué puede alcanzar el pipeline

Dar al pipeline la credencial del migrator traslada un gran poder a un sistema en el que mucha gente puede influir. Conviene ser explícito al respecto, porque es donde este diseño puede torcerse sin hacer ruido.

Qué puede hacer el pipeline

Un job que ejecuta `runMigration` puede crear, alterar o eliminar cualquier objeto del esquema, cambiar cualquier grant sobre objetos que posee el propietario, y cambiar las dos tablas append-only, `ledger_entries` y `audit_events`. Ningún rol de aplicación ni ningún rol humano puede editar filas de esas tablas, incluido `human_breakglass`; la única excepción estructural es que el propietario, y por tanto una migración en ejecución, puede alterarlas. El control para esa excepción no es un privilegio. Es la revisión.

Así que la pregunta sobre el pipeline es: ¿quién puede hacer que ejecute algo? Eso incluye a quienes hacen merge en la rama de deploy, a quienes pueden editar la definición del pipeline, a quienes pueden añadir un paso a un job que tiene la credencial y a cualquiera que pueda leer el entorno del job mientras se ejecuta.

Acotar el alcance

Acotaría ese alcance de cinco maneras, aproximadamente en orden de cuánto quita cada una.

1. **Solo un job tiene la credencial.** El job de migración del entorno de producción es el único lugar donde se puede emitir el login del migrator, y lo emite la identidad del job, no una variable que pueda leer cualquier job. Los builds de ramas sin revisar, los jobs de prueba y los pipelines de pull request nunca la reciben.
2. **La definición del pipeline se revisa como las migraciones.** Un cambio en el job de migración, en sus permisos o en las reglas de rama que lo activan es un cambio en el alcance del propietario, y necesita al mismo revisor que comprobaría una migración.
3. **El disparador es una rama protegida.** Hacer merge en ella exige la revisión de una persona que no sea quien la escribió, que es el control four-eyes de Ledgerly para los cambios de esquema. Para las migracio-

nes que tocan grants, propiedad o las tablas append-only, el revisor debe pertenecer al equipo de plataforma.

4. **La credencial es de vida corta.** Un job que falla y deja un log con un secret deja un secret que ya ha vencido.
5. **Cada ejecución queda registrada.** El pipeline escribe lo que aplicó, con qué login, desde qué commit, en un almacén que el pipeline no puede reescribir, y escribe la entrada de inicio antes de que se ejecute la migración. El Capítulo 14 cubre dónde vive ese almacén.

La revisión misma puede ser en parte mecánica. Una pequeña comprobación puede leer la migración antes que una persona y señalar las sentencias que necesitan una mirada más atenta:

```
function classifyMigration(migration, lane):
  flags = []
  refusals = []
  for each statement in migration.statements:
    if statement.touches("ledger_entries") or
statement.touches("audit_events"):
      add(flags, "append-only table: platform review required")
      if statement.kind in ["GRANT", "REVOKE", "CHANGE OWNER",
"CHANGE DEFAULT PRIVILEGES", "CREATE
VIEW"]:
        add(flags, "changes access: platform review required")
        if statement.kind in ["CREATE TABLE", "ADD COLUMN"] and
(statement.table is new or statement.table in
sensitiveTables) and
not hasExplicitGrants(migration, statement.table):
          add(flags, "new structure without a grant decision")
        if statement.kind in ["INSERT", "UPDATE", "DELETE"]:
          add(flags, "data change in a migration: backfill rule
applies")
        if statement.kind in lockHeavyKinds:
          add(flags, "long lock: scheduled window and a named
watcher")
        if statement.kind in ["DROP TABLE", "DROP COLUMN"] and not
migration.isContractStep:
          add(flags, "destructive change outside a contract step")
        if lane == "expedited" and statement.kind not in
expeditedAllowlist:
          add(refusals, "not allowed in the expedited lane: use the
normal lane")
  return flags, refusals
```

En el lane normal la comprobación no decide nada. Hace que la atención del revisor caiga sobre las líneas que cambian el acceso, mantienen un bloqueo o destruyen datos, y hace visible la omisión de un grant. Una com-

probación que bloquea un merge por cada señal, sin forma de darse por enterado de una, será esquivada en un mes, así que una señal exige un aprobador designado y no prohíbe el cambio. El lane expedited es el único lugar donde la comprobación sí rechaza: una lista `refusals` no vacía hace fallar el job antes de que se ejecute, y ningún aprobador puede anularlo. La sección siguiente explica por qué.

El lane normal no rechaza nada de forma tajante, y eso es una decisión: una lista rígida de sentencias prohibidas sería una segunda política junto a la matriz `deny`, `gate` y `allow` del Capítulo 9. Eliminar `ledger_entries` queda señalado dos veces, como cambio `append-only` y como cambio destructivo, y el equipo de plataforma tiene que decir que sí.

Por qué no una persona con un ticket

Algunos equipos prefieren que un DBA aplique cada migración a mano después de la aprobación. Para una base de datos pequeña con uno o dos cambios de esquema al mes puede ser una elección razonable, y la cuestión del alcance del pipeline desaparece. El costo aparece cuando la persona no está, cuando el cambio debe aplicarse de forma idéntica en varios entornos, o cuando el paso manual se aleja de lo que se revisó. La debilidad del pipeline es la amplitud de su alcance; la debilidad de la ruta manual es que nada obliga a que lo aplicado sea igual a lo aprobado. Me inclino por el pipeline con los límites de arriba, y revisaría la elección si la lista de acceso del propio pipeline resultara imposible de mantener corta.

Los bloqueos y el riesgo de disponibilidad

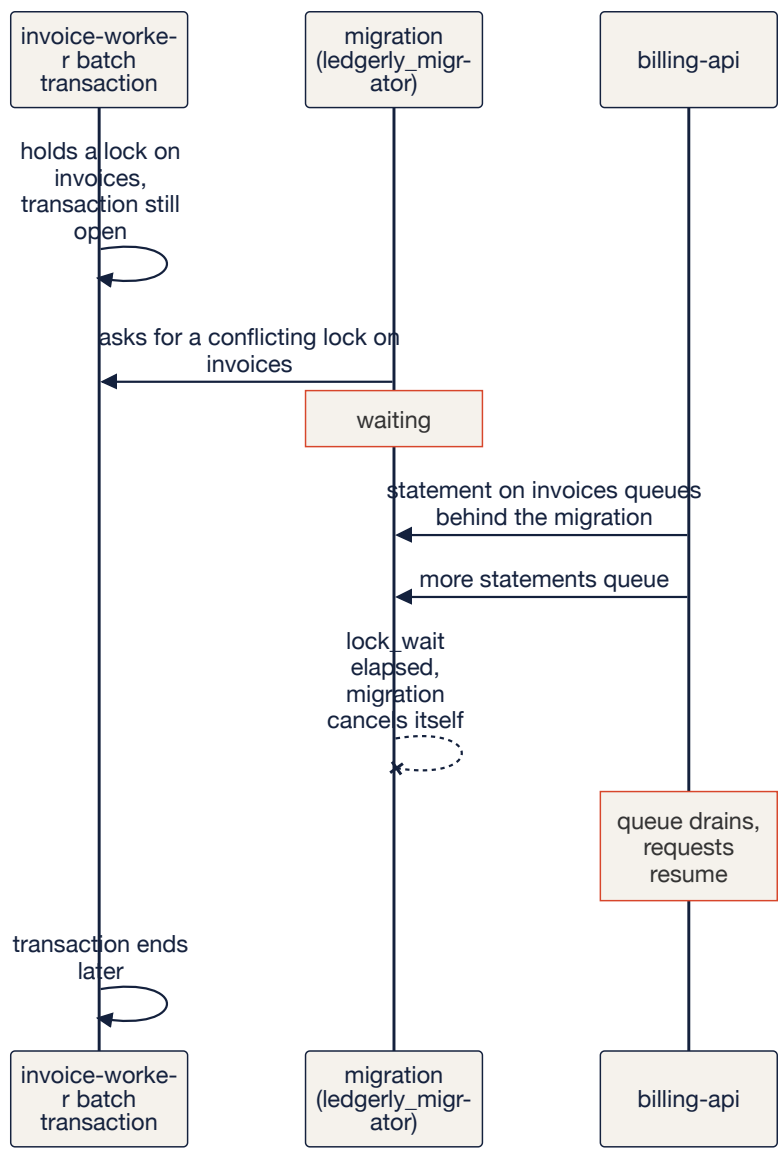
Volvamos ahora a los noventa segundos de Ledgerly. El mecanismo que hay detrás es común a todos los motores, y la defensa se construye con un puñado de hábitos.

Cómo una migración en espera cierra una tabla

Una migración que altera `invoices` necesita un bloqueo que entra en conflicto con las lecturas y escrituras ordinarias. Solo puede tomar ese bloqueo cuando ya no queda abierta ninguna transacción que tenga un bloqueo en conflicto. Si hay una transacción larga abierta, la migración espera. Mientras espera, las sentencias nuevas sobre la tabla quedan en cola

detrás de la solicitud pendiente de la migración, porque dejar que se le adelanten la dejaría sin servicio. La tabla queda entonces efectivamente cerrada hasta que termina la transacción antigua o la migración se rinde.

La secuencia de abajo muestra el orden de los hechos. Busca el punto en el que la espera de bloqueo da a la migración una salida antes de que la cola crezca.



Lo importante es la autocancelación. Sin una espera de bloqueo, la migración espera todo el tiempo que siga abierta esa transacción del batch, y `billing-api` hace cola durante todo ese tiempo. Con una espera de bloqueo corta, la migración falla a los pocos segundos, la cola se vacía y el pipeline puede intentarlo de nuevo más tarde.

Una espera de bloqueo corta y un retry paciente

El Capítulo 5 fijó `lock_wait 3s` y ningún `statement timeout` para `ledgerly_migrator`. Ese emparejamiento es deliberado. Un cambio de esquema puede necesitar mucho tiempo para hacer su trabajo, pero debe esperar muy poco para empezar. El `retry` pertenece entonces al pipeline:

```
function applyWithRetry(migration, maxAttempts, pauseSeconds):
  attempt = 1
  while attempt <= maxAttempts:
    try:
      runMigration(migration)
      return
    on LockWaitExceeded:
      // a clean no-op only if nothing had committed: one
      transactional
      // unit, or a locking step that comes first
      if not migration.retrySafe:
        notify("platform", "partial migration, not retried: "
+ migration.id)
        raise MigrationBlocked
        notify("platform", "migration waiting on a lock: " +
migration.id)
        sleep(pauseSeconds * attempt)
        attempt = attempt + 1
      // the table is busy for longer than the retry window; a person
      decides
      notify("platform", "migration blocked after retries: " +
migration.id)
      raise MigrationBlocked
```

Unas observaciones. Una espera de bloqueo fallida es un no-op limpio solo cuando todavía no se ha confirmado nada, y por eso una migración lleva una marca `retrySafe`: es una sola unidad transaccional, o su paso de bloqueo va primero. Una migración de varios pasos en un motor donde cada sentencia DDL hace commit por sí sola no se reintenta a ciegas, porque el segundo intento se encontraría con el trabajo del primero. Se detiene y una persona decide. La pausa crece con el número de intentos para que los intentos repetidos no se conviertan ellos mismos en un patrón de car-

ga. Y el estado final, `MigrationBlocked`, es una cuestión humana: averiguar qué transacción se interpone y decidir si esperar, cancelarla o reprogramar. Esa decisión implica matar la sesión de alguien, que es una cuestión de acceso en sí misma y pertenece al Capítulo 15 cuando ocurre durante un incidente.

Sentencias largas y cambios en línea

Algunas operaciones mantienen un bloqueo mientras dure su ejecución, como reescribir una tabla grande para cambiar el tipo de una columna. Para esas, la espera de bloqueo no es el problema; lo es la duración. Los motores difieren en qué sentencias son rápidas, y la respuesta cambia con la versión del motor, así que comprueba cada una. Las herramientas de cambio de esquema en línea y las construcciones de índice sin bloqueo evitan el bloqueo largo, pero crean y eliminan objetos, de modo que necesitan el mismo alcance de propietario que cualquier migración y se ejecutan como el migrator, dentro del pipeline. `classifyMigration` marca estas sentencias como de bloqueo largo, y una migración de bloqueo largo recibe una ventana programada y una persona con nombre que vigila, escritas en la descripción de la migración. Eso no es un fallo de la automatización; algunos cambios son eventos.

Expand and contract

El cambio de esquema más seguro es aquel con el que pueden convivir el código antiguo y el nuevo de la aplicación. El patrón para esto es **expand and contract**: cambiar el esquema en pasos aditivos, pasar la aplicación al nuevo y solo entonces eliminar lo que ya no se usa. Importa para este libro porque cada paso es una migración pequeña con un bloqueo pequeño, y porque mantiene los deploys reversibles.

Considera un cambio hipotético en Ledgerly, usado solo para mostrar la forma: renombrar `total_minor` de `invoices` a `amount_minor`. El esquema canónico conserva `total_minor`, y el ejemplo supone una variante en la que la columna se renombra. Los pasos son así:

1. **Expand.** Una migración añade `amount_minor` como columna nullable, con un bloqueo breve. El código antiguo la ignora.

2. **Escribir en ambas.** Las versiones nuevas de `billing-api` e `invoice-worker` escriben la columna nueva además de la antigua, de modo que las versiones antiguas y nuevas pueden ejecutarse en paralelo.
3. **Backfill.** Las filas antiguas reciben valores para la columna nueva, como se describe más abajo.
4. **Leer la columna nueva.** Si la versión se comporta mal, la columna antigua sigue siendo correcta, así que el rollback es un deploy y no una reparación.
5. **Contract.** Tras un período en el que nada lee ni escribe `total_minor`, una migración final la elimina. Este es el único paso destructivo, y tiene su propio gate: se revisa como destructivo y nunca va en la versión que introdujo la columna nueva.

Expand and contract cuesta tiempo y atención. Un cambio que podría ser una sola migración se convierte en cuatro o cinco, repartidas en varios deploys, y durante los pasos intermedios la aplicación lleva código para ambas formas. Para un equipo pequeño con un solo servicio y una ventana de mantenimiento, una única migración puede ser la mejor opción. Para Ledgerly, con cinco servicios que leen las mismas tablas y ninguna ventana que el negocio acepte, el patrón es lo que mantiene corto el bloqueo y barato el rollback.

Backfills dentro de la migración, y correcciones de datos fuera de ella

El paso 3 es un cambio masivo de datos, y la misma regla se aplicaría a una división del total en neto e impuesto, con una diferencia: la tabla antigua no contiene ninguna cifra de impuesto, así que ese backfill no se podría calcular y tendría que dejar vacías las columnas nuevas hasta que una corrección de datos las llenara. El Capítulo 10 define una corrección de datos como un cambio en datos de producción hecho por una persona fuera de la ruta normal de la aplicación, para reparar un estado que la aplicación no puede reparar, así que el libro necesita aquí una línea: un backfill que pertenece a un cambio de esquema se ejecuta dentro de la migración como `ledgerly_migrator`; cualquier otro cambio masivo de datos es una corrección de datos y sigue el Capítulo 10.

¿Qué hace que un backfill pertenezca al cambio de esquema?

- Su único propósito es dar a la estructura nueva sus primeros valores, mediante una regla enunciada en la migración.
- Está descrito en la migración, se revisa con ella y lo aplica el pipeline.
- Cada valor que escribe podría recalcularse a partir de datos que ya están en la tabla.

Si cualquiera de estas condiciones es falsa, es una corrección de datos. El renombrado muestra dónde cae la línea: cada valor escrito en `amount_minor` es el `total_minor` de la misma fila, una regla que la migración enuncia y que un revisor puede comprobar. Supón que el equipo se da cuenta, mientras hace el backfill, de que algunas facturas antiguas tienen un `total_minor` incorrecto. Corregir esos totales no forma parte del cambio de esquema. Sigue el Capítulo 10, con un conteo de las filas afectadas, una salvaguarda sobre el conteo de filas esperado y un segundo revisor, porque una copia que preserva un valor erróneo sigue siendo un backfill correcto. El poder del migrator es grande, y no debe convertirse en la ruta de todo cambio incómodo.

En una tabla grande, el backfill se ejecuta en lotes, cada uno una transacción corta sobre un conjunto acotado de filas. Se pausa cuando crece el retraso de la réplica (Capítulo 6), se rinde de forma visible si el retraso no se despeja, y puede detenerse y reiniciarse sin perder trabajo.

La misma regla se aplica a las dos tablas append-only. Una migración puede añadir una columna a `ledger_entries`, pero no reescribe las entradas existentes para llenar esa columna con valores corregidos; una corrección es una entrada nueva, emitida mediante la ruta de corrección de `payments-service`. Una migración que sí toque esas tablas la señala el paso de clasificación de arriba y pasa al equipo de plataforma para su revisión.

DDL de hotfix y drift

Todo equipo se enfrenta tarde o temprano a la noche en que un índice arreglaría una consulta lenta y el pipeline tarda, digamos, veinte minutos. La pregunta es qué dice el modelo de acceso sobre el atajo.

No hay atajo por consola, así que construye un lane rápido

En el estado objetivo nadie tiene el login del propietario.

`human_breakglass` escribe `INSERT` y `UPDATE` y no lleva DDL, y el super-usuario es un instrumento sellado que no se usa en el trabajo diario (Capítulo 3). Así que una persona en el primary no tiene una forma legítima de añadir un índice a mano. Es deliberado, y traslada la pregunta al pipeline: ¿con qué rapidez puede ejecutar un cambio revisado cuando la necesidad es urgente?

Darí­a al pipeline un **lane expedited** en lugar de un bypass. El lane usa el mismo job, la misma credencial y el mismo registro; se diferencia en el momento, no en quién debe aprobar. El four-eyes se mantiene: un revisor que no es quien la escribió sigue aprobando, y quien la escribe no hace merge por su cuenta. Lo que se relaja es el período de espera programado y, para una lista corta de clases de sentencia en allowlist, el requisito del equipo de plataforma, con una revisión de seguimiento por parte del equipo de plataforma el siguiente día laborable. `classifyMigration` hace cumplir la allowlist: construir un índice, eliminar un índice y añadir una columna nullable son elegibles. Una sentencia que cambia grants, propiedad o una vista, toca una tabla append-only o destruye datos no lo es, porque esos son los cambios para los que existe la revisión de plataforma, y una migración que contenga uno se rechaza en el lane y pasa por el normal. El Capítulo 15 describe el acceso durante los incidentes; lo que importa aquí es que la urgencia cambia la velocidad del proceso, no la identidad que ejecuta el DDL ni la revisión que recibe una sentencia peligrosa.

El costo es que el lane expedited es una opción permanente de saltarse algo de cautela, y una opción así se usa. Registra con qué frecuencia se usa, y trata un número creciente como señal de que el lane normal es demasiado lento, no de que el equipo se haya vuelto menos cuidadoso.

Drift

El **drift** es cualquier diferencia entre el esquema que describen las migraciones y el esquema que existe en producción. Entra por un hotfix aplicado a mano, por una migración que se editó después de aplicarse, por un cambio a nivel de propietario hecho por una herramienta que nadie recor-

daba y por grants alterados fuera de la definición de acceso. El drift en el esquema sorprende a la siguiente migración. El drift en los grants es un cambio de acceso que ninguna revisión vio.

Una comprobación programada puede encontrar ambos tipos:

```
function detectDrift(environment):
    expected = buildExpectedSchema(appliedMigrations(environment))
    actual = readLiveSchema(environment)

    findings = []
    for each difference in compare(expected, actual):
        add(findings, difference)
    for each difference in compare(expectedGrants(),
readLiveGrants(environment)):
        add(findings, difference)

    if length(findings) > 0:
        notify("platform", "schema or grant drift in " + environment)
        recordCheckRun(environment, findings)

function checkDriftCheckAge(environment):
    // scheduled separately, so a stopped drift job cannot hide itself
    if now() - lastSuccessfulRun("detectDrift", environment) >
maxRunAge:
        notify("platform", "drift check has not completed recently")
```

Dos cosas la hacen útil. El estado esperado se deriva de las migraciones y de la definición de acceso del Capítulo 3, de modo que hay exactamente una fuente de verdad con la que comparar. Y la comprobación cubre los grants, no solo las tablas, porque un grant que aparece fuera de cualquier archivo es el tipo de drift más importante para este libro. Cuando se encuentra una diferencia, la resolución es una de dos: escribir una migración que registre el cambio, si debe quedarse, o escribir una que lo revierta, si no. Resolver el drift editando el registro de lo que se aplicó oculta el cambio en lugar de revisarlo.

Nota: No “arregles” el drift aplicando una sentencia correctiva directamente como superusuario o como propietario. La sentencia que elimina la diferencia es en sí misma un cambio sin revisar en producción. Ponla en una migración y deja que pase por el lane.

Tablas nuevas, columnas nuevas y los grants que les siguen

El Capítulo 3 eligió un valor por defecto vacío para las tablas nuevas: una tabla no es legible por nadie salvo el propietario hasta que una migración enuncia un grant. Esta sección es la otra mitad de esa decisión, porque es en las migraciones donde la decisión se mantiene o se olvida.

Los grants viven en la migración

Cada migración que crea una tabla enuncia sus grants en el mismo archivo. Quien revisa ve entonces todo el efecto de acceso de un cambio en un solo diff, que es el argumento a favor del valor por defecto vacío. Para una tabla nueva, `invoice_adjustments` (una tabla ilustrativa, que no forma parte del esquema canónico), el archivo es así. Su columna de texto libre `reason` se clasifica como Confidential para el peor caso, igual que `invoice_lines.description`, de modo que los lectores distintos del propietario de la tabla reciben grants a nivel de columna que la dejan fuera:

```
CREATE TABLE invoice_adjustments (  
  id          text primary key,  
  invoice_id  text not null,  
  reason      text not null,  
  delta_minor integer not null,  
  created_at  timestamp not null  
);  
  
// the migration states every grant it means to give  
GRANT SELECT, INSERT ON invoice_adjustments TO app_billing_api;  
// reason is Confidential free text: left out of the broad read grants  
GRANT SELECT (id, invoice_id, delta_minor, created_at) ON  
invoice_adjustments TO app_reporting_job;  
GRANT SELECT (id, invoice_id, delta_minor, created_at) ON  
invoice_adjustments TO human_readonly;
```

Olvidar un grant tiene dos resultados posibles, y la diferencia es la razón del valor por defecto vacío. Con un valor por defecto vacío, el servicio que necesita la tabla falla en su primera consulta. Es ruidoso, se descubre en una prueba o en el primer deploy, y se arregla con una migración de una línea. Con un privilegio por defecto amplio, un grant olvidado es silencioso en la dirección contraria: la tabla siguiente es legible por quien nombre el valor por defecto, y nadie lo nota porque nada falla. Un fallo ruidoso es más barato que una exposición silenciosa.

El fallo solo es ruidoso si algo ejecuta la consulta. Por eso incluiría, en el pipeline, un paso de smoke posterior a la migración que compruebe que cada rol de aplicación puede usar las tablas que debe usar. Los roles de aplicación no tienen login, así que el paso necesita un mecanismo. Usaría un login de prueba que sea miembro de cada rol de aplicación y cambie a cada uno por turno, donde el motor lo admita; la alternativa es un login de prueba por rol. Cualquiera de las dos opciones amplía el alcance del pipeline, porque un login que puede convertirse en cualquier rol de servicio vale más para un atacante que la credencial de un solo servicio. Por eso lo emite la workload identity del job, vence con el job y se usa solo para este paso. A cambio, el paso atrapa el grant olvidado antes de que lo haga el rollout del servicio.

Columnas nuevas en tablas que ya tienen grants

Una columna nueva se comporta de manera distinta según cómo esté otorgada su tabla.

Estilo de grant de la tabla	Una columna nueva es...	Consecuencia
Grant a nivel de tabla	legible por todos los roles que tienen el grant	Una columna sensible nueva queda expuesta de inmediato
Grant a nivel de columna	legible por nadie hasta que se nombra	Una columna nueva debe otorgarse para poder usarse

Para las tablas con columnas sensibles, como `customers`, `bank_accounts` y `payment_tokens`, eso aboga por grants a nivel de columna donde el motor los admita, que es la elección que hizo el Capítulo 3. La columna nueva empieza entonces cerrada, y la migración que la añade es donde el equipo decide qué roles pueden leerla y si es Confidential o Restricted (el Capítulo 12 cubre la clasificación). Una migración que añade una columna a una de esas tablas sin ninguna sentencia de grant es una invitación a hacer una pregunta, y la comprobación de clasificación puede hacerla.

Donde el motor carece de grants a nivel de columna, se aplica la alternativa del Capítulo 3: las columnas Restricted viven en tablas separadas como `bank_account_secrets` y `payment_token_values`. Una columna Restricted nueva pertenece a una de esas tablas, nunca a la tabla principal, porque los grants de la tabla principal son a nivel de tabla y el techo del acce-

so JIT se traza a su alrededor. Es fácil violarlo por accidente, ya que añadir una columna a la tabla que todos ya conocen es lo natural, y es el tipo de error que solo notará un revisor que conozca la alternativa.

Vistas sobre tablas sensibles

En muchos motores una vista se ejecuta con los derechos de su propietario, lo que significa que una vista sobre una tabla con columnas Restricted puede revelar lo que los roles propios de sus lectores no podrían. Por eso `classifyMigration` señala `CREATE VIEW` como un cambio de acceso, por eso la migración que crea una pasa por el equipo de plataforma, y por eso una vista sobre una tabla sensible se juzga como un cambio de acceso y no como una comodidad.

Trade-offs y costos

El pipeline es un objetivo de alto valor. Una credencial que puede ejecutar DDL de nivel propietario, en manos de un sistema que mucha gente puede editar, es una concentración de poder. Un pipeline revisado no impide que alguien con alcance de nivel propietario haga algo sin revisar; hace que ese algo sea visible después del hecho, que es un control más débil de lo que suena. Los límites de este capítulo reducen el riesgo sin eliminarlo. Si la lista de acceso de tu pipeline es larga y nadie es responsable de ella, arregla eso antes de apoyarte en cualquier otra cosa.

Las revisiones añaden retraso y valen lo que valga quien revisa. Una comprobación four-eyes en las migraciones ralentiza los cambios pequeños y puede volverse una formalidad, sobre todo cuando quien revisa ve, digamos, diez migraciones a la semana y todas son una columna. La comprobación de clasificación ayuda al señalar las líneas peligrosas. No sustituye a la atención. Una revisión que siempre se aprueba en segundos es la señal que hay que vigilar, y el Capítulo 8 desarrolla el argumento completo.

Expand and contract multiplica el número de cambios. Un cambio lógico se convierte en varios deploys. Para un equipo con una ventana de mantenimiento, un servicio y una tabla pequeña, puede que no valga la pena. El patrón compensa cuando muchos componentes leen la tabla y nadie puede detenerlos mientras dura un cambio.

Una espera de bloqueo corta hace fallar las migraciones. Es el comportamiento buscado, y es un costo. En una tabla ocupada una migración puede necesitar varios intentos, y alguien debe decidir qué ocurre después del último. Si fijas la espera demasiado corta para tu carga de trabajo, el pipeline enviará un page al equipo de plataforma por ruido. Ajústala a partir de cómo son tus propias transacciones largas, y déjalo dicho en el runbook.

Las comprobaciones de drift producen hallazgos que alguien debe asumir. Una comprobación que alerta a un canal que nadie lee no encuentra nada. Decide quién es responsable de cada clase de hallazgo antes de activar la comprobación.

Cuándo no usar este diseño. Para una base de datos desechable, un prototipo o un proyecto de una sola persona sin datos de clientes, un migrator con un rol de propietario y un pipeline revisado es más de lo que pide la situación. Ejecuta las migraciones desde la máquina de un desarrollador, mantén la base de datos de producción fuera de la ruta crítica y añade el pipeline cuando alguien más que tú dependa de los datos. El diseño de este capítulo es para el momento en que el esquema lo comparten varios servicios y un cambio equivocado le cuesta algo a los clientes.

Errores comunes

- **Dejar que el servicio cree o altere sus propias tablas al arrancar.** Quita un paso del deploy y da al servicio la propiedad, que incluye el poder de eliminar y volver a otorgar. Mantén desactivada en producción la gestión automática del esquema, y ejecuta las migraciones como un paso aparte con el migrator.
- **Guardar la credencial del migrator en el gestor de contraseñas del equipo “por si hay una emergencia”.** Es tentador porque el pipeline a veces falla en el peor momento. Convierte a cada persona que la tiene en propietario y elimina la propiedad de que cada uso de la propiedad corresponda a una ejecución del pipeline. Construye en su lugar el lane expedited.
- **Ejecutar una migración sin espera de bloqueo.** La migración espera tanto como la transacción más antigua que tiene delante, y todas las

sentencias que vienen detrás esperan también. Da al migrator una espera de bloqueo corta y pon el retry en el pipeline.

- **Esconder una corrección de datos dentro de un backfill.** Parece eficiente corregir valores malos mientras la tabla está abierta de todos modos. La revisión de la migración se diseñó para la definición de una columna, no para un cambio en los importes de los clientes. Si el cambio no es un primer llenado de la estructura nueva mediante una regla enunciada, es una corrección de datos y sigue el Capítulo 10.
- **Eliminar la columna antigua en la misma versión que la nueva.** Ahorra una migración y elimina el rollback. Mantén el paso de contract separado, diferido y revisado como destructivo.
- **Depender de los privilegios por defecto para que las tablas nuevas funcionen sin una línea de grant.** Un valor por defecto amplio convierte un grant olvidado en una exposición silenciosa. Mantén el valor por defecto vacío y deja que cada migración enuncie sus grants.
- **Añadir una columna Restricted a una tabla principal en un motor sin grants a nivel de columna.** Los grants de la tabla son a nivel de tabla y el techo JIT se construye a su alrededor. Pon las columnas Restricted en las tablas de secrets separadas.
- **Tratar un hallazgo de drift como ruido.** Un grant que existe en producción y en ningún lugar del código es una decisión de acceso que nadie revisó. Resuélvelo con una migración que lo registre o lo revierta.
- **Editar una migración después de aplicarla.** Los entornos que ejecutaron la versión antigua ahora difieren de los que ejecutan la nueva. Añade en su lugar una migración nueva.

Ideas clave

- El DDL se ejecuta como `ledgerly_migrator` desde el pipeline, como propietario durante el tiempo de una migración, y ningún rol de aplicación tiene privilegios de DDL.
- El migrator tiene una autoridad permanente de propietario, ejercida mediante una credencial de vida corta: la tiene un solo job, la emite la workload identity, cada ejecución se registra antes de empezar y las personas nunca la usan.

- Como el pipeline puede alcanzar el poder del propietario, su propia lista de acceso y su definición se revisan como una migración; las migraciones que tocan grants o las tablas append-only pasan al equipo de plataforma, y el lane expedited acepta solo una allowlist de clases de sentencia que hace cumplir `classifyMigration`.
- Una espera de bloqueo corta con un retry en el pipeline evita que una migración bloqueada cierre una tabla, y las herramientas en línea y los pasos aditivos se ocupan de los cambios que de otro modo mantendrían un bloqueo durante mucho tiempo.
- Expand and contract mantiene funcionando juntos el código antiguo y el nuevo y convierte el rollback en un deploy, al costo de más migraciones; un backfill que llena la estructura nueva mediante una regla enunciada se ejecuta dentro de la migración, y cualquier otro cambio masivo de datos es una corrección de datos y sigue el Capítulo 10.
- El drift en el esquema o en los grants se encuentra comparando el estado vivo con lo que describen las migraciones y la definición de acceso, y se resuelve con una migración, no con una edición silenciosa.
- Cada migración enuncia los grants de lo que crea y mantiene vacíos los valores por defecto, de modo que un grant olvidado falle con ruido en lugar de exponer una tabla en silencio.

Los cambios de esquema son el único lugar donde un pipeline, y no una persona, tiene el mayor poder sobre los datos. La parte siguiente pasa de quién puede llegar a la base de datos a lo que hay dentro de ella: el Capítulo 12 pregunta qué columnas merecen protección, cómo clasificarlas y cómo enmascararlas o retenerlas sin romper el trabajo que las necesita.

PARTE IV

Proteger los datos

Capítulo 12 — Datos sensibles: clasificación, enmascaramiento y minimización

Imagina un lunes en Ledgerly. Un cliente escribe a soporte para decir que un pago no ha llegado y pregunta si la cuenta bancaria registrada es la correcta. Sam, de soporte, abre la página del cliente en `admin-console` y encuentra el número de cuenta completo en pantalla. Leerlo en el chat es la forma más rápida de responder, y una captura en el ticket es la forma más rápida de pasar el caso a un colega. Nadie en la historia hace nada irrazonable, y al final de la tarde el número vive en un log de chat y en un adjunto de ticket.

Dos escritorios más allá, Jordan, del equipo de datos, está armando un informe sobre `ledgerly_analytics` y se da cuenta de que la tabla `customers` de esa copia lleva nombres de contacto, emails y teléfonos que ningún informe necesita. Robin, que está de on-call, lee un log de la aplicación para averiguar por qué falló un recordatorio, y encuentra la dirección de email del destinatario dentro del mensaje de error. Cada exposición es pequeña, y las tres comparten una forma: los datos llegaron a un lugar donde nadie había decidido que debían estar.

Este capítulo añade un paso del estado objetivo: las columnas sensibles se clasifican y, por defecto, se enmascaran o se retienen, y la copia de analítica no contiene datos Restricted sin tratar. La pregunta que responde es qué necesita ver realmente cada persona y cada copia, y cómo construir el sistema para que muestre eso y nada más.

En el modelo de cinco partes de este libro (**Identidad, Grant, Ruta, Ejecución, Registro**), el capítulo trabaja sobre dos partes. El Grant decide qué columnas puede leer un rol, y la Ejecución decide en qué forma vuelve un valor a través de la ruta. Nada de esto depende de cómo llegó una credencial a una persona; un login perfectamente protegido que lee un número de cuenta completo lo ha expuesto igual.

El problema y por qué es difícil

Los controles de los capítulos anteriores deciden quién llega a qué tabla y a qué columnas. No saben por qué importa una columna. Un grant sobre `customers` trata igual un indicador de estado y un email de contacto, y el esquema llama `text` a ambos. Nada en la definición de una columna dice que `invoice_lines.description` es texto libre que una persona puede haber llenado con un nombre, ni que `iban` es el tipo de valor cuya exposición no se deshace rotando algo.

La segunda dificultad es que las personas necesitan una parte de los datos, y una regla que diga que nadie puede verlos produce rodeos. Soporte debe responder preguntas sobre las cuentas de pago. El on-call debe averiguar por qué falló un recordatorio. El equipo de datos debe contar clientes por segmento. Cuando la ruta autorizada no muestra nada, aparecen las no autorizadas: capturas de pantalla, exports ad hoc, un grant permanente “hasta que termine el proyecto”. La pregunta útil no es si una persona puede ver un valor, sino qué forma del valor necesita la tarea: el valor completo, sus últimos dígitos, un sí o un no, o un conteo.

La tercera dificultad es que los datos no se quedan en la tabla. Se copian a un almacén de analítica, se escriben en líneas de log, se repiten en mensajes de error, se pegan en tickets y se envían en exports. Todo control sobre el primary se detiene en el borde de la base de datos, y el valor sigue su camino. Por eso la respuesta tiene que estar construida dentro del sistema y no puede quedarse en una política. Una política que dice “trata los datos personales con cuidado” no puede ejecutarla un pipeline de migraciones, una carga programada ni un formateador de logs. Las decisiones tienen que escribirse en las cosas que se ejecutan: el esquema, los grants, la carga y el código de logging.

Los regímenes de protección de datos y de tarjetas de pago, como GDPR, KVKK o PCI DSS, suelen plantear una familia de preguntas: qué datos personales tienes, por qué, adónde van, quién puede verlos y durante cuánto tiempo. Una clasificación guardada en el esquema es la forma en que puedes responderlas. Tu equipo legal y de cumplimiento es responsable de la interpretación de cualquier régimen; este capítulo solo describe la ingeniería que hace que las respuestas se puedan encontrar. El catálogo es eviden-

cia de lo que hace el sistema, no una afirmación de que cumpla algún régimen; el Capítulo 14 vuelve sobre qué evidencia puede recoger una revisión de accesos.

Clasificación que vive en el esquema

La **clasificación de datos** ordena los datos en clases con nombre, y cada clase decide el tratamiento. Ledgerly usa cuatro.

Clase	Qué significa	Ejemplos en Ledgerly	Tratamiento por defecto
Restric- ted	Su exposición da- ñaría a los clien- tes o a la empresa de formas difíciles de deshacer	iban, account_holder_name, token_value	Retenido para human_readonly; nunca en una solicitud just-in- time (JIT); se cambia solo me- diante una ruta de la aplicación; ausente de la copia de analítica
Con- fi- den- tial	Datos persona- les o co- merciales sobre clientes y contactos	contact_email, billing_address, provider_reference, invoice_lines.description	Enmascarado o retenido para human_readonly; sin enmascarar solo en un SELECT JIT con un aprobador; enmascarado o elimi- nado en la copia de analítica, sal- vo las pistas de visualización que existen para mostrarse (provider, expires_on; véase la tabla de decisiones más abajo)
In- ter- nal	Datos operati- vos sin contenido personal	amount_minor, status, due_on, account_code	Legible por los roles que lo nece- sitán; copiado a analítica
Pu- blic	Seguro de publicar	Ninguno entre las ocho tablas	Solo integridad

La fila Public está vacía a propósito. Existe para que la pregunta “¿esto se puede publicar?” tenga una respuesta distinta del silencio.

Cuatro reglas hacen utilizables las clases. Son lo que escribiría antes de elegir cualquier herramienta.

- **Clasifica columnas, no tablas.** `customers` contiene un indicador de estado y un número de teléfono, y los dos necesitan tratamientos distintos. Solo a efectos de inventario, una tabla toma la clase más alta entre sus columnas.
- **Clasifica el texto libre pensando en el peor caso.** Una columna en la que la gente escribe acabará conteniendo lo que la gente escriba. `invoice_lines.description` es Confidential porque puede contener datos personales, y ningún filtro puede decir qué filas los contienen.
- **Un valor derivado hereda la clase de su origen hasta que alguien argumente lo contrario.** Un hash con clave de un `iban` es Restricted hasta que una persona haya argumentado que no lo es. `payment_tokens.last_four` es Confidential y no Restricted porque existe para mostrarse, y esa decisión se tomó una vez y quedó por escrito.
- **Lo desconocido es Confidential.** Una columna que nadie ha clasificado se trata como Confidential y se marca como provisional, para que el trabajo pendiente sea visible y el valor por defecto sea el cerrado.

Un catálogo en el mismo repositorio que las migraciones

Una clasificación que vive en una wiki sufre drift, porque nada de lo que cambia el esquema lee la wiki. La mantendría como un archivo en el repositorio que contiene las migraciones, y la cambiaría en el mismo pull request que añade la columna. Algunos motores también permiten etiquetas o comentarios en las columnas; cualquiera de las dos formas sirve mientras el pipeline pueda leerla.

```
// illustrative catalog entry, one per column
table column_classification:
  table_name          text not null
  column_name         text not null
  class               text not null    // Restricted,
Confidential, Internal, Public
  reason              text not null    // one sentence, for the
reviewer
  owner               text not null    // a team, never a person
  human_readonly      text not null    // readable, masked,
withheld
  analytics_copy       text not null    // copy, transform, drop
  reviewed_on         date null        // null means provisional
```

El catálogo hace posibles dos comprobaciones. La primera se ejecuta contra cada migración, junto a los indicadores descritos en el Capítulo 11. A diferencia de esos indicadores, que piden a un revisor que mire, esta bloquea el merge, porque clasificar una columna cuesta una línea y una línea ausente es justo lo que todo el control existe para detectar.

```
function checkColumnClassification(migration, catalog):
    problems = []
    for each column in addedColumns(migration):
        entry = catalog.find(column.table, column.name)
        if entry is null:
            add(problems, "unclassified: " + column.table + "." +
column.name)
            continue
        if entry.class == "Restricted" and not
inRestrictedHome(column):
            // a main table whose grants are table-level cannot
withhold it
            add(problems, "Restricted outside its home: " +
column.name)
            if entry.class == "Restricted" and entry.analytics_copy !=
"drop":
                add(problems, "Restricted column would reach analytics")
        if problems is not empty:
            raise MergeBlocked(problems)
```

Aquí `inRestrictedHome` designa una tabla donde los grants pueden retener la columna: una tabla principal con grants a nivel de columna, o las tablas de secrets separadas del Capítulo 3 en los motores que no los tienen. Cuando quien añade la columna no está seguro, la salida no es saltarse la comprobación sino clasificar la columna como Confidential con `reviewed_on` vacío, y dejar que la siguiente revisión de accesos lo resuelva.

La segunda comprobación se ejecuta de forma programada y compara el esquema vivo con el catálogo, igual que el Capítulo 11 compara el esquema vivo con las migraciones. Una columna creada a mano en una mala noche aparece como un hallazgo con responsable, y el hallazgo se resuelve con una migración y una línea de catálogo, no con una edición silenciosa.

Minimización: los datos que no guardas

La **minimización** consiste en no recoger, copiar ni conservar lo que nadie necesita. Es el control más barato de este capítulo, porque un valor que nunca se almacenó no tiene máscara que mantener, ni copia que encontrar, ni línea que redactar. Todos los demás controles de aquí suponen que el dato existe; este pregunta si debería existir.

Para cada columna sensible, haría cuatro preguntas en este orden.

1. ¿Alguna funcionalidad u obligación necesita este valor? Si la respuesta es “podría ser útil”, la respuesta es no.
2. ¿Podríamos guardar un sustituto? Ledgerly guarda `payment_tokens`, tokens emitidos por el proveedor que representan instrumentos de pago, y no los instrumentos mismos. Esa sola decisión elimina del esquema toda una clase de valores.
3. ¿Podríamos guardar menos? Una pista de visualización como `last_four` permite a soporte y a los clientes reconocer un instrumento sin el instrumento.
4. ¿Podríamos guardarlo durante menos tiempo? Los datos de contacto de un cliente cerrado tienen una finalidad hasta que se salda la última factura y pasan los plazos de disputa; después se conservan solo durante un período corto y definido, y luego se eliminan.

Dos decisiones sobre dónde viven los datos importan más que cualquier regla de retención. La primera es que **las tablas append-only no deberían llevar datos personales**. `ledger_entries` contiene `amount_minor` y `account_code`, ambos Internal, y por eso una corrección puede ser una entrada nueva: nada en la tabla necesita borrarse jamás para atender una solicitud sobre una persona. `audit_events` se refiere a su sujeto mediante un `subject_id` opaco por la misma razón. La segunda es el texto libre. Un campo estructurado con un conjunto fijo de valores se puede clasificar y enmascarar; un cuadro de notas no es ninguna de las dos cosas, así que, donde una funcionalidad pueda usar un código, un estado o una referencia de producto en lugar de una frase, elegiría eso.

La retención es un job, no una frase en una política. Algo tiene que ejecutarse de forma programada, encontrar las filas que han pasado su período y eliminarlas o anonimizarlas, y ese algo necesita sus propios privilegios

acotados, sus propias entradas de auditoría y un responsable. No sé cuál es el período correcto para tus datos, y este libro tampoco. Depende de lo que exijan tus contratos, tus reguladores y tu propio proceso de disputas, y tu equipo legal tiene que responderlo antes de que una persona de ingeniería escriba el job. Lo que sí puede decir la ingeniería es que los backups, los exports y la copia de analítica también contienen datos antiguos, y una eliminación que solo alcanza al primary no ha terminado. El Capítulo 13 vuelve sobre esto.

Enmascaramiento: dinámico y estático

El **enmascaramiento** reemplaza u oculta un valor para que el real no sea visible mientras la forma sigue siendo utilizable. La **redaction** es la versión burda: el valor se elimina de una salida o de un log. Los dos tipos de enmascaramiento difieren en dónde actúan.

	Enmascaramiento dinámico	Enmascaramiento estático
Qué cambia	Lo que devuelve una consulta	La copia misma
Cuándo	En el momento de la lectura, por lector	En el momento de la carga, una vez
El valor real sigue almacenado	Sí, en el mismo almacén	No en la copia
Uso típico	<code>human_readonly</code> en una réplica	La copia de analítica, los datos de staging
Cómo falla	Comportamiento del motor, filtros, grants directos	Una copia desactualizada o recortada de más

Enmascaramiento dinámico para personas

`human_readonly` es el valor por defecto con el que empieza toda sesión, así que su enmascaramiento es el que más gente encuentra. La notación de abajo es ilustrativa, porque los motores expresan el enmascaramiento de formas muy distintas: algunos ofrecen una máscara en la columna, otros un privilegio de desenmascarar, otros una vista fija por tabla.

```
// illustrative notation. The grant lists the columns the role may
read;
// each mask rule changes what a read through the role returns
GRANT SELECT (id, created_at, status, contact_name, contact_email,
               contact_phone, billing_address) ON customers TO
human_readonly;

MASK RULE contact_name_initials  ON customers (contact_name)
  FOR human_readonly USING initials();
MASK RULE contact_email_partial  ON customers (contact_email)
  FOR human_readonly USING keep_first(1);
MASK RULE contact_phone_tail     ON customers (contact_phone)
  FOR human_readonly USING keep_last(2);
MASK RULE billing_address_hidden ON customers (billing_address)
  FOR human_readonly USING redact();
MASK RULE provider_reference_tail ON payments (provider_reference)
  FOR human_readonly USING keep_last(4);

// free text has no shape to mask, so it is withheld: not in the
column list
// (illustrative column list)
GRANT SELECT (id, invoice_id, created_at) ON invoice_lines TO
human_readonly;
```

La elección de la función depende de para qué sirve la columna. Una técnica no es mejor ni peor en sí misma; cada una tiene su lugar y su forma de fallar.

Técnica	Qué obtiene quien lee	Conviene a	Dónde deja de ayudar
Parcial	Los primeros o los últimos caracteres	Una pantalla, un resultado pegado	Los valores cortos muestran casi todo de sí mismos
Redaction	Un marcador fijo o nada	Direcciones, texto libre	No permite confirmar nada
Seudónimo con clave	Un sustituto estable	Joins entre cargas	Rotar la clave rompe los joins
Generalización	Un valor más amplio, como un país o un mes	Analítica	Las columnas combinadas pueden singularizar a una persona (Capítulo 13)

Un hash simple no está en la lista. Un hash de un valor tomado de un conjunto tan pequeño que se pueden probar todos los candidatos, como un número de teléfono, se puede revertir probándolos, así que unseudónimo necesita una clave guardada fuera de la base de datos.

No toda columna Confidential se enmascara. Una columna que existe para mostrarse sigue siendo legible: `payment_tokens.provider`, `last_four` y `expires_on` son las pistas de visualización que permiten a una persona reconocer un instrumento, y enmascararlas solo empujaría a la gente hacia el valor Restricted que sustituyen. Los campos de contacto, la dirección de facturación y `provider_reference` identifican a una persona o una transacción y se enmascaran. `invoice_lines.description` se retiene.

De ahí sale la regla de la que depende el Capítulo 8. `human_readonly` ya lee, así que una tripleta `SELECT JIT` solo hace falta para una columna Confidential que el rol enmascara o retiene, y lo que añade es esa columna sin enmascarar. Las columnas Restricted nunca aparecen en una tripleta JIT, así que aquí nada cambia para ellas. La solicitud y la pantalla del aprobador deberían llevar, por tanto, una línea explícita, “Columnas sin enmascarar: `contact_email`”, separada de la lista de tablas, para que el aprobador vea qué máscaras elimina la solicitud y no solo qué tablas toca.

Advertencia: Comprueba cómo resuelve tu motor un login que lee una columna a través de `human_readonly` y además tiene un grant directo sobre la columna. Si gana la máscara, un `SELECT JIT` no muestra nada nuevo y la aprobación no significa nada. Si ganan los valores sin tratar para todos los lectores, la máscara es decoración. Prueba ambos casos en una copia antes de confiar en cualquiera de los dos.

Dos límites van junto a la técnica. Primero, en muchos motores una máscara se aplica a lo que devuelve una consulta, no a lo que comparan sus condiciones, así que un lector que puede filtrar por una columna enmascarada puede aprender algo sobre sus valores mediante conteos y coincidencias. Trata el enmascaramiento dinámico como protección contra la exposición casual, como una pantalla, una captura o un resultado pegado. Segundo, enmascarar una salida no hace nada por el valor almacenado, que permanece en todos los backups, réplicas y dumps del primary.

El primer límite da una regla de decisión para la inferencia. Pregunta cuán barato le resultaría a un lector recuperar el valor filtrando: un puñado de sondeos de igualdad sobre `contact_email` es barato, y un conteo de clientes por un prefijo de teléfono enmascarado también lo es. Donde es

barato y el lector no debe conocer el valor, retén la columna en lugar de enmascararla, o niega los predicados sobre la columna enmascarada para ese rol. Donde es caro, como sondear un valor libre largo un carácter a la vez mediante miles de sentencias registradas, el enmascaramiento más el registro de sentencias es un control razonable.

Enmascaramiento en el flujo de resultados

Algunas rutas de acceso no pueden confiar en que el motor enmascare. Un gateway que se sitúa entre la persona y la base de datos puede aplicar reglas al propio flujo de resultados, antes de que las filas lleguen a la pantalla o a un archivo de resultados. Eso ayuda donde el motor no tiene funcionalidad de enmascaramiento, y las reglas pueden ser las mismas para todos los motores que haya detrás.

Una regla de flujo depende de saber de qué columna vino un valor, y un alias como `SELECT api_key AS k` derrota a las reglas que coinciden por nombre de columna. Las herramientas que enmascaran resultados, Query-Proxy entre ellas, añaden reglas que coinciden también por el valor, lo cual es una segunda capa, nunca un sustituto del grant. En el momento de escribir esto, documenta por sí misma esta limitación de las reglas basadas en nombres, y otras herramientas siguen enfoques similares. La consecuencia para Ledgerly es que una máscara de flujo va detrás de los grants de columna del Capítulo 3, y una columna que un rol nunca debe ver se retiene mediante el grant, no se deja a un patrón.

Nota: En los motores sin grants a nivel de columna, Ledgerly usa la alternativa del Capítulo 3: las columnas Restricted están en `bank_account_secrets` y `payment_token_values`. Enmascarar `human_readonly` suele significar entonces una vista enmascarada fija por tabla, creada una vez por una migración y propiedad de `ledgerly_owner`, con el rol teniendo `SELECT` sobre la vista y no sobre la tabla base. Un grant JIT a nivel de tabla sobre la tabla base muestra sin tratar todas las columnas que no son Restricted, las Confidential incluidas, y el aprobador ha aceptado exactamente eso. Es una vista fija, no una vista creada por solicitud, y es un mecanismo de enmascaramiento para columnas Confidential. No es una segunda alternativa para las Restricted: la única alternativa para esas son las tablas de secrets.

Enmascaramiento estático para la copia de analítica

La copia de analítica es donde el enmascaramiento es más barato y más útil, porque el valor real puede dejarse fuera de la copia por completo. El informe de Jordan necesita segmentos de clientes, conteos e importes. No necesita nombres, emails ni teléfonos, así que para la mayoría de las columnas Confidential el tratamiento correcto es `drop`, y una transformación queda para las pocas que los analistas usan de forma demostrable.

La carga es un job programado, no un servicio, y se ejecuta con su propio login y el rol `app_analytics_load`, como cualquier otro principal en la revisión de accesos. El rol tiene `SELECT` solo sobre las columnas que el catálogo no marca como Restricted, así que el grant es la primera protección y el catálogo la segunda. No son independientes, porque el grant se deriva de las marcas Restricted del catálogo: una columna mal clasificada como Internal pasa las dos. Lo que añade el grant es que una columna marcada como Restricted no se puede leer aunque una entrada de transformación sea incorrecta, y la comprobación independiente es la aserción de esquema de `verifyMasked`, que compara la copia con el plan. Dentro de ese grant, la carga decide columna por columna a partir del catálogo, y falla cerrado.

```

function loadAnalyticsTable(table, catalog):
    plan = []
    for each column in sourceColumns(table):
        entry = catalog.find(table, column)
        if entry is null:
            // an unclassified column never travels; the load says so
            notify("platform", "not loaded, unclassified: " + table +
"." + column)
            continue
        if entry.class == "Restricted" and entry.analytics_copy !=
"drop":
            raise CatalogInvalid
            if entry.analytics_copy != "drop":
                add(plan, { column: column, treatment:
entry.analytics_copy })
            // build beside the live table, check, then swap: a failure leaves
the
            // previous load in place, so the copy never holds a half-masked
table
            next = table + "_next"
            writeMasked(next, from: table, plan: plan)
            if not verifyMasked(next, plan):
                dropTable(next)
                raise MaskingCheckFailed
            swapTable(next, into: table)

```

`verifyMasked` hace dos comprobaciones. Afirma el esquema: el conjunto de columnas del destino debe ser igual al plan, y no puede existir en él ninguna columna Restricted ni eliminada. Después toma una muestra de filas y falla si una columna transformada sigue siendo igual a su valor de origen. La aserción de esquema detecta una columna que llegó por una vía que el plan no listaba; la muestra detecta una transformación mal configurada. Ninguna es una prueba de que la copia no contenga nada sensible, porque la muestra solo lee algunas filas.

El precio de fallar cerrado es visible: cuando un desarrollador añade una columna y olvida el catálogo, la comprobación de migraciones debería haberlo detenido, y si no lo hizo, la tabla de analítica se carga sin esa columna y el equipo de datos nota un hueco. Lo aceptaría. Un informe al que le falta una columna es un ticket; una copia con una columna sin enmascarar es una exposición que nadie nota.

Un seudónimo con clave para joins merece una decisión propia. Permite a los analistas seguir a un cliente a lo largo de varias cargas sin saber quién es, y solo funciona mientras la clave se mantenga fuera del alcance del equipo de datos y no cambie. Una rotación de la clave cambia todos los

seudónimos, de modo que el historial deja de unirse. Para muchos equipos el `customer_id` opaco ya cumple esa función, y un segundo sustituto añade una clave que proteger sin ningún beneficio.

Tokenización y cifrado a nivel de aplicación

La **tokenización** reemplaza un valor por un sustituto que solo un servicio controlado puede volver a mapear. No es enmascaramiento, porque el mapeo existe, y normalmente no es cifrado, porque el sustituto normalmente no se calcula a partir del valor en absoluto. Los esquemas que sí lo calculan a partir del valor con una clave son cifrado con otro nombre, y exigen el mismo cuidado con las claves. Ledgerly ya se beneficia de ella en un punto: `token_value` es un token del proveedor, el proveedor guarda el mapeo, y `payments-service` es el único que llama. La base de datos de Ledgerly, por tanto, nunca guarda el instrumento.

Construir un segundo servicio de tokenización para los números de cuenta bancaria es un paso mayor. Añade a la ruta de pagos un sistema que debe estar disponible cuando lo estén los pagos, necesita una lista de quienes pueden volver a mapear un token, y cada mapeo es un evento que alguien debería poder revisar. No lo construiría al tamaño de Ledgerly. Restringir las columnas, restringir el único servicio que las lee y cifrar los valores daría la mayor parte de la protección con menos cosas que mantener en marcha. Si un socio regulado o un proveedor de pagos lo exige, la exigencia decide.

El **cifrado a nivel de aplicación** significa que el servicio cifra un valor antes de que la base de datos lo vea, de modo que los roles, las réplicas, los backups y los dumps contienen todos texto cifrado. Esta es su fortaleza distintiva: protege el valor en las copias de datos a las que un grant no llega (Capítulo 13). El cifrado que hace la capa de almacenamiento, en disco o en un snapshot, protege contra un volumen perdido y no hace nada contra una consulta, y los equipos a menudo lo confunden con control de acceso. El cifrado a nivel de aplicación se parece más a una frontera real, y es un compromiso más pesado.

- **La clave decide quién puede leer.** Los datos están tan protegidos como su clave. Da permiso de descifrado a la workload identity de `payments-service` y a nadie más, de modo que la decisión sea una

pregunta sobre quién puede descifrar y a través de qué. Las jerarquías de claves, la rotación y el backup de claves son una disciplina aparte, y la documentación del servicio de claves las cubre mejor que este libro.

- **La columna pierde sus funciones de consulta.** No puedes buscar, ordenar, filtrar por rango ni imponer unicidad sobre texto cifrado de la forma habitual, así que elige columnas que solo se lean de vuelta por identificador.
- **Un servicio que puede descifrar cuando se le pide, puede recibir esa solicitud.** El cifrado traslada la frontera al servicio, así que quienes la llaman, su ritmo y sus entradas de auditoría pasan a ser el control, y la entrada de auditoría se escribe antes del descifrado, como se ve a continuación.

```
function readPayoutAccount(accountId, onBehalfOf):
    row = selectBankAccount(accountId)          // the stored value is
    ciphertext
    // the audit entry comes first: if the log is unreachable, nothing
    is decrypted
    try:
        auditLogAppend({ actor: "payments-service", action: "decrypt",
                        subject: row.id, on_behalf_of: onBehalfOf })
    on AuditLogUnavailable:
        raise DecryptRefused
    // the key service checks the caller's workload identity, not the
    request
    return keyServiceDecrypt(row.iban, keyId: row.key_id)
```

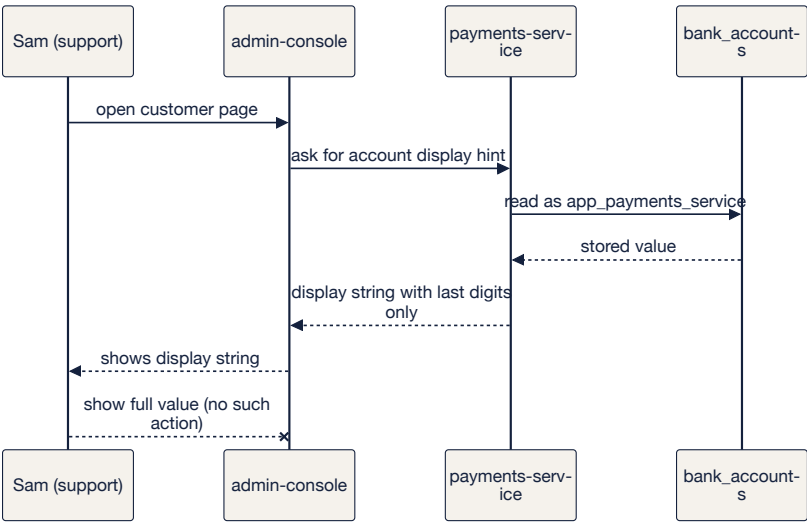
La matriz no cambia si cifras. `human_breakglass` sigue leyendo las columnas `Restricted`, pero lo que lee es texto cifrado, y descifrar requiere un permiso de clave que el rol no tiene. Decidiría de antemano si un incidente que necesita el texto en claro puede pedir acceso a la clave, con qué aprobador, y lo escribiría en el runbook de `break-glass` del Capítulo 8. Dejar esa decisión para el incidente es la manera de improvisar una regla.

Trabajar sin ver: el caso de soporte

Volvamos a Sam. El objetivo es que Sam pueda responder al cliente por completo y no vea nunca el valor entero. La mayoría de las preguntas de soporte resultan necesitar menos que el valor.

Lo que so- porte nece- sita hacer	Lo que necesita ver	Cómo se le proporciona
Decir al cliente qué cuenta está registrada	Los últimos dígitos	<code>payments-service</code> devuelve una cadena de visualización; o la pista se guarda cuando se escribe la cuenta, como <code>last_four</code> en el caso de los tokens
Comprobar que quien escribe es el contacto	Un sí o un no	Una comprobación acotada en el servicio que ya tiene el grant, que compara lo que dijo el cliente con el valor almacenado y responde “coincide” o “no coincide”
Revisar las facturas del cliente	Estados, importes, fechas de vencimiento, solo de este cliente	<code>app_admin_console</code> lee las columnas Internal de <code>invoices</code> , y la regla de filas sobre el cliente del caso, del Capítulo 3, acota las filas
Corregir un número de cuenta equivocado	Nada, y no a mano	La ruta de la aplicación: <code>admin-console</code> se lo pide a <code>payments-service</code> , una segunda persona aprueba y el cambio queda registrado en <code>audit_events</code>

El diagrama muestra la primera fila. Mira la última flecha: una acción que no existe es un control más fuerte que una que existe y se rechaza.



No construiría un botón de “revelar” al principio. Un revelar que pide un motivo y registra el clic suena razonable, y en un mes hacer clic es un hábito y su log es algo que nadie lee. Si aparece una necesidad real, conviértela en una solicitud con un aprobador, como hace el Capítulo 8 con cual-

quier lectura Confidential sin enmascarar, y cuenta con qué frecuencia se usa. Una funcionalidad que se usa a diario te está diciendo que a la forma de visualización le falta algo.

Quien esté de on-call sigue la misma regla bajo otra presión. Robin, depurando un recordatorio fallido, necesita saber que una dirección de destinatario fue rechazada, y pocas veces necesita la dirección. Cuando Robin sí necesita el `contact_email` sin enmascarar, eso es un `SELECT JIT` que nombra la columna, con un aprobador que lee la lista de columnas. Las lecturas de on-call nunca se aprueban automáticamente: una regla que omite las columnas Confidential aprueba una solicitud que no concede nada, y una regla que las incluye elimina el único control que tiene sentido. El costo es una espera a una mala hora, y la manera de reducirlo es eliminar la necesidad del valor en los casos comunes, lo que cubre la siguiente sección.

Datos sensibles en logs y mensajes de error

Los logs son la copia de datos que nadie diseña. Los escriben muchos componentes, los leen muchas personas, se conservan mucho tiempo y se consultan con herramientas que nunca formaron parte del modelo de acceso. Cuatro lugares concentran la mayor parte de las fugas.

- **Logs de la aplicación y mensajes de error.** Un mensaje construido a partir de un valor, como “no se pudo enviar a la dirección”, pone el valor en el log, en el rastreador de errores y a veces en la pantalla de un cliente. Los drivers y los ORM también pueden registrar parámetros enlazados cuando el logging de depuración está activo.
- **Logs de la base de datos.** El registro de sentencias puede guardar literales, y un log de consultas lentas conserva el texto de una consulta junto con los valores que contiene.
- **Registros de sentencias.** El access gateway registra cada sentencia que envía una persona (Capítulo 7), y una persona que escribe un valor en una condición ha puesto ese valor en el registro.
- **Tickets y chat.** Un resultado pegado para explicar un problema es una copia con el público equivocado y sin fecha de fin.

Advertencia: Un log de sentencias contiene literales, de modo que el registro de auditoría del Capítulo 14 y los registros de sentencias del gateway son, a su vez, almacenes de datos sensibles. Registra las sentencias como fingerprints con los literales eliminados mediante redaction, clasifica el almacén al menos como Confidential, restringe quién lo lee y dale un período de retención; de lo contrario, el control que vigila los datos se convierte en una copia más de ellos. El mecanismo corresponde al Capítulo 14.

Mi regla preferida para el logging de la aplicación es una allowlist, no un depurador. Un depurador busca patrones que se parecen a valores sensibles y los elimina; detecta las formas en que pensó su autor y deja pasar el resto. Una allowlist nombra los campos que pueden aparecer y descarta todo lo demás, de modo que un campo nuevo permanece en silencio hasta que alguien decide registrarlo.

```
function buildLogEvent(event, allowedFields):
  safe = {}
  dropped = 0
  for each field in event.fields:
    if field.name in allowedFields:
      safe[field.name] = field.value
    else:
      dropped = dropped + 1
  // the count is the signal: a field dropped every day is either a
  // missing allowlist entry or a value somebody tried to log
  safe["dropped_field_count"] = dropped
  return safe
```

Los mensajes de error siguen la misma disciplina. Un error lleva un código e identificadores, como un id de factura, un id de cliente y un estado de entrega, y no el valor que falló. Así es también como el caso del recordatorio de Robin deja de necesitar la dirección real: si el registro del fallo contiene un estado como “destinatario rechazado” y el id de la factura, el log responde la pregunta sin la dirección, y el registro del gateway sobre quién leyó la dirección, si alguien lo hizo, queda en un solo lugar.

La redaction en el pipeline de logs es una red de seguridad, y se le escaparán cosas. La comprobación que no me saltaría es un canario en las pruebas: ejecuta la aplicación con un marcador de forma reconocible en cada campo sensible y haz fallar el build si aparece en cualquier salida de log.

Prueba la allowlist donde importa y cuesta unas pocas líneas. La depuración se vuelve más lenta cuando el log contiene identificadores en lugar de valores, y el `dropped_field_count` es adonde van a parar los mensajes de “¿por qué no puedo ver esto?”, en lugar de que un desarrollador vuelva a activar el logging de parámetros en producción.

Las decisiones de columnas de Ledgerly

La tabla de abajo reúne el capítulo. Es el registro de decisiones de las columnas sensibles de las ocho tablas, en la forma en que puede leerla una revisión de accesos. En esa columna, “Retenida” y “Sin acceso” significan ambas que el rol no tiene ningún grant sobre la columna, de modo que una consulta que la nombra es rechazada. La columna `human_readonly` indica el valor por defecto de una sesión; la columna JIT indica si una solicitud puede nombrar la columna.

Columna	Clase	human_readonly	SELECT JIT	Copia de analítica
bank_accounts.iban, account_holder_name	Restric- ted	Sin acceso	Nunca	Eliminada
payment_tokens.token_value	Res- tric- ted	Retenida	Nunca	Eliminada
payment_tokens.provider, last_four, expires_on	Con- fi- den- tial	Legible	No hace falta	Se conservan provider y expires_on, se elimina last_four
customers.contact_name, contact_email, contact_phone	Con- fi- den- tial	Enmascarada	Nombra la co- lumna, sin enmas- carar	Eliminada
customers.billing_address	Con- fi- den- tial	Con redaction	Nombra la co- lumna, sin enmas- carar	Eliminada

Columna	Clase	human_readonly	SELECT JIT	Copia de analítica
payments.provider_reference	Confidencial	Últimos cuatro caracteres	Nombra la columna, sin enmascarar	Eliminada
invoice_lines.description	Confidencial	Retenida	Nombra la columna, sin enmascarar	Eliminada
amount_minor, currency, status, due_on, account_code	Internal	Legible donde la matriz lo concede	No hace falta	Copiada

Dos entradas merecen un comentario. Las columnas Restricted se eliminan de la copia de analítica y nunca se ofrecen a un rol humano distinto de `human_breakglass`; la única ruta que las cambia es la ruta de la aplicación, con una segunda persona que aprueba (el Capítulo 9 explica cómo la política de ejecución encamina allí las operaciones con gate). La misma ruta se ocupa del vínculo con el propietario: `customer_id` en `bank_accounts` y `payment_tokens` queda dentro del techo JIT porque no es Restricted, pero mover una fila a otro cliente cambia de quién es la cuenta, así que un cambio del vínculo con el propietario tiene gate y se encamina a la ruta de producto, y nunca se hace como una corrección de datos ordinaria. Y donde el motor no tiene grants a nivel de columna, la misma tabla sigue valiendo, con un cambio: las columnas Restricted viven en las tablas de secrets, la columna JIT dice “a nivel de tabla” y la pantalla de aprobación lo dice con palabras.

Trade-offs y costos

La clasificación es mantenimiento. Un catálogo que es correcto el día en que se escribe empieza a envejecer de inmediato. Alguien tiene que ser responsable de cada decisión de clase, revisar las provisionales y respon-

der la pregunta que plantea una columna sin clasificar. Si nadie revisa el catálogo, se convierte en el tipo de registro que está completo y equivocado, que es peor que un hueco honesto.

El enmascaramiento cuesta tiempo de depuración. Robin no puede ver el email real, así que un bug que depende de sus caracteres exactos necesita una solicitud JIT y un aprobador. Acepto ese costo a cambio del acceso que cierra este capítulo. No es gratis, y un equipo que lo paga todas las noches empezará a buscar la manera de rodearlo, así que mide con qué frecuencia se solicita la lectura sin enmascarar y arregla la causa.

El enmascaramiento dinámico depende del motor. El comportamiento ante filtros, grants directos y vistas difiere, y una máscara que no has probado en tu motor es una esperanza. La variante basada en vistas añade objetos que las migraciones deben mantener sincronizados con las tablas base.

Fallar cerrado produce huecos. La carga de analítica que rechaza las columnas sin clasificar deja al equipo de datos sin una columna hasta que alguien la clasifica. Nombra quién es responsable de esa espera, y mantén la revisión de clasificación lo bastante ágil como para que el hueco sea de horas y no de semanas.

El cifrado a nivel de aplicación quita funciones y añade una dependencia. El servicio de claves queda en la ruta de cada lectura de la columna, y su caída es una caída de pagos. Como la entrada de auditoría va primero, un registro de auditoría inalcanzable también detiene esas lecturas, lo cual es un acoplamiento deliberado y una dependencia de disponibilidad más. La pérdida de funciones de consulta empuja la búsqueda hacia otras columnas, y el backup de la clave es una cosa más que hay que probar.

Algunos controles de aquí pueden convertirse en teatro de seguridad. Un botón de desenmascarar que siempre se pulsa, un catálogo que nadie revisa, un depurador de logs que nadie prueba y una columna enmascarada que cualquier lector puede sondear con filtros añaden fricción o papeleo y no reducen nada.

Cuándo no usar todo esto. Un sistema pequeño con un puñado de tablas y sin datos personales de clientes no necesita un archivo de catálogo, una carga enmascarada ni un servicio de claves. Una lista de una página con

las columnas sensibles, grants a nivel de columna que las dejen fuera y una allowlist de logs dan la mayor parte del valor. Añade enmascaramiento cuando un segundo público necesite los datos, añade cifrado cuando te preocupe una copia a la que no se puede rodear con grants, y deja por escrito qué evento te haría pasar al siguiente paso.

Errores comunes

- **Tratar el cifrado en reposo como control de acceso.** Es tentador porque es un solo ajuste y satisface una lista de verificación. Protege contra un disco perdido y no hace nada contra una consulta. Mantén los grants y usa cifrado a nivel de aplicación donde una copia necesite protección.
- **Enmascarar en el cliente y no en la base de datos.** Una máscara en la interfaz web es fácil de construir, y una conexión directa, un export o una respuesta de API la eluden. Enmascara donde los datos salen del almacén.
- **Clasificar tablas y no columnas.** Una etiqueta por tabla protege de más el indicador de estado y protege de menos el teléfono. Clasifica cada columna y deriva la etiqueta de la tabla a partir de ellas.
- **Poner Internal por defecto en las columnas nuevas.** Internal es la clase de menor fricción, así que se convierte en la clase de todo lo que no se ha revisado. Haz de Confidential el valor por defecto y retíralo mediante revisión.
- **Enmascarar texto libre con patrones.** Un patrón encuentra los números y los emails que esperaba su autor y deja pasar un nombre escrito en una nota. Retén las columnas de texto libre a los lectores amplios.
- **Usar un hash sin clave como seudónimo.** Parece irreversible. Para un valor con pocas posibilidades se puede revertir probándolas. Usa un seudónimo con clave que se mantenga fuera de la base de datos, o elimina la columna.
- **Copiar producción a staging “y enmascararla después”.** La ventana entre la copia y la máscara es una copia de todo. Enmascara por el camino, como hace la carga de analítica, y trata staging como una de las puertas del Capítulo 13.

- **Añadir una columna Restricted a una solicitud JIT para ahorrar un paso.** El techo la rechaza por una razón, y la tentación es más fuerte de noche. Los valores Restricted solo cambian por la ruta de la aplicación con aprobación four-eyes, y en los motores sin grants a nivel de columna están en las tablas de secrets.
- **Aprobar automáticamente las lecturas de on-call.** Elimina la pausa en la que una persona comprueba qué columna Confidential se está desenmascarando. Mantén al aprobador y reduce el número de veces que se necesita el valor.
- **Registrar los cuerpos de las solicitudes en el nivel error.** El cuerpo es lo más útil para un desarrollador y contiene todos los campos que escribió un cliente. Registra identificadores y estados, y deja que una allowlist decida qué más aparece.

Ideas clave

- Clasifica cada columna, trata el texto libre pensando en el peor caso y haz que las columnas sin clasificar sean Confidential por defecto, para que los datos nuevos empiecen cerrados y el trabajo de abrirlos sea visible.
- Mantén el catálogo de clasificación en el repositorio que contiene las migraciones y bloquea un merge que añada una columna sin clase, porque la comprobación cuesta una línea y su ausencia es el fallo.
- La minimización va primero: los datos que nunca se almacenaron no necesitan máscara, y los datos personales que se mantienen fuera de las tablas append-only no pueden convertirse en un problema de borrado.
- El enmascaramiento dinámico conviene a `human_readonly` y protege contra la exposición casual; el enmascaramiento estático conviene a la copia de analítica, donde el valor real puede eliminarse por completo y la carga falla cerrado ante cualquier cosa sin clasificar.
- Da a las personas la forma del valor que necesita su tarea, como una cadena de visualización o un sí o un no, y mantén las columnas Restricted fuera de toda solicitud JIT y de toda escritura en SQL directo.
- Una máscara en el flujo de resultados es una segunda capa detrás del grant, nunca un sustituto, y un rol de carga que no puede leer las co-

lumnas Restricted hace que un error del catálogo falle cerrado.

- Los logs, los mensajes de error y los registros de sentencias son copias de datos: registra identificadores mediante una allowlist, mantén los valores fuera de los mensajes y clasifica el propio registro de auditoría.

Los controles de este capítulo protegen los datos mientras están en la base de datos y mientras una consulta los devuelve. El siguiente capítulo sigue a los datos hacia fuera, a los backups, los dumps, los exports y las copias de staging, donde ninguno de estos roles se aplica.

Capítulo 13 — Backups, exports y otras puertas laterales

Imagina una tarde de viernes en Ledgerly. Robin persigue un bug en los recordatorios de facturas que solo aparece con datos realistas: facturas con fechas de vencimiento inusuales, clientes con varios contactos, descripciones de líneas con texto libre que nadie había previsto. Staging tiene una docena de clientes hechos a mano, y el bug no se manifiesta en ninguno. Alguien recuerda que hay un dump antiguo de producción en el almacenamiento compartido, y Alex, que se encarga de las bases de datos, lo restaura en staging para que todo el equipo pueda mirar.

El bug aparece en menos de una hora. El dump se queda en la carpeta, y la base de datos restaurada se queda en staging. Staging no tiene access gateway, porque no es producción. Su login es compartido, todos los ingenieros pueden llegar a él, los jobs de CI se ejecutan contra él, y un contratista que se incorporó el mes pasado tiene una cuenta. `invoice-worker` en staging usa los mismos ajustes de email saliente que producción, y su job de recordatorios está activado.

Nadie decidió exponer datos de clientes. Cada paso fue una respuesta sensata al problema de una tarde, y los controles construidos en los capítulos anteriores protegen una base de datos que ya no contiene la única copia de su contenido.

Este capítulo añade el último paso del estado objetivo antes de la Parte V: los backups, los exports y las copias de staging se tratan como puertas con responsables y vencimiento. La pregunta es esta: una vez que los datos salen del primary a propósito, para recuperación, para un proveedor, para una prueba o para una persona que necesita mirar, ¿cómo mantienes los controles con los datos y no con la base de datos de la que vinieron?

En el modelo de cinco partes (**Identidad, Grant, Ruta, Ejecución, Registro**), una copia es un lugar donde el Grant y la Ruta ya no se aplican, porque los datos salieron por una vía que no lleva ninguno de los dos. El tra-

bajo del capítulo es lograr que la copia misma lleve un Grant: una lista de lectores, un responsable y una fecha de fin, con un Registro de quién la hizo y quién la eliminó.

El problema y por qué es difícil

El Capítulo 6 definió una **puerta lateral** como una copia de datos o una ruta de export que elude los controles de acceso del primary. Las réplicas y la copia de analítica son puertas que construyes deliberadamente y puedes nombrar. Este capítulo trata del resto: backups y snapshots, dumps, archivos CSV, bases de datos de staging, unos pocos miles de filas en un laptop, un cache de dashboard, un archivo en la bandeja de entrada de un proveedor. La mayoría se hicieron por una buena razón. Pocas tienen responsable.

La dificultad central es que los controles de acceso viven en un sistema, no en los datos. El primary tiene un access gateway, un login por sesión, grants a nivel de columna, timeouts y un registro de auditoría. Un dump del primary no tiene nada de eso, porque un dump es un archivo. Quien pueda leer el archivo tiene todo lo que contiene, incluidas las columnas que el esquema de roles retiene a todos los roles humanos salvo `human_breakglass`. Un backup que contiene las columnas Restricted es, para las personas que pueden leerlo, un grant permanente sobre esas columnas que ningún gateway emitió y ninguna revisión lista.

La respuesta obvia es una política: no copiar datos de producción. Falla por una razón simple. Copiar es como se hace el trabajo. La recuperación necesita una copia por definición. Un proveedor necesita archivos. Un bug que depende de la forma de los datos reales necesita algo que parezca real. Una política que lo prohíbe todo se rodea, y las copias que resultan son las no gestionadas. La mejor pregunta es qué copias hacen falta, quién es responsable de cada una, quién puede leerla y cuándo termina.

Una segunda dificultad es que estas puertas las hacen equipos distintos por razones distintas, así que ningún responsable único las ve todas. El equipo de plataforma es responsable de los backups, el equipo de datos de la copia de analítica, el equipo de backend de staging, finanzas de la relación con el proveedor, y cada ingeniero del contenido de su propio laptop.

Un inventario es el primer control, y se encuentra en buckets de almacenamiento, calendarios de jobs y configuración, no en un documento de política.

Un mapa de las puertas laterales

Antes de los controles, un mapa. La tabla lista las puertas que cubre este capítulo, qué contiene cada una en el caso de Ledgerly y quién llega a ella por defecto si nadie ha diseñado otra cosa.

Puerta	Qué contiene	Quién llega a ella por defecto	Primer control
Backups y snapshots de volumen	Todo, incluidas las columnas Restricted y las tablas de secrets	Cualquiera con acceso al almacenamiento o a la infraestructura	Poderes separados para crear, restaurar y eliminar; cifrado con su propia clave
Dumps en almacenamiento compartido	Todo, como un archivo legible	Cualquiera que pueda abrir la carpeta	Encontrar, asignar un responsable, eliminar o mover bajo las reglas de backup
Exports humanos (CSV, tablas pegadas)	Lo que devolvió la consulta	La persona, y luego quien reciba el archivo	Aprobación por niveles, vencimiento, sin datos Restricted
Bases de datos de staging y de pruebas	Una copia de producción, o algo que se le parece	Todos los ingenieros, CI, contratistas	Datos enmascarados o sintéticos, mensajes salientes desactivados
Laptops	Conjuntos de resultados, bases de datos locales, archivos guardados	La persona usuaria del laptop y sus backups	Eliminar la necesidad; muestras enmascaradas en su lugar
Herramientas de BI	Extractos en cache, informes programados, dashboards compartidos	Todos con quienes se comparte la herramienta	Leer solo la copia de analítica, identidad por persona cuando sea posible
Archivos para proveedores	Exports de facturas y del libro mayor	El personal y los sistemas del proveedor	Rol acotado, lista de columnas, retención, vencimiento del archivo

Todas las filas tienen la misma forma de problema: una copia de datos con los controles de un sistema de archivos o de un producto SaaS, y una lista de lectores que nadie escribió. El registro del Capítulo 6 ya existe por esta

razón, y lo extendería a cada fila de arriba. Una puerta que no está en el registro no es una puerta que alguien vaya a revisar.

```
table data_copy:                                // extended from Chapter 6
  name      text primary key                    // "staging_refresh_current"
  kind      text not null                      // backup, dump, export,
  staging, laptop,                               // bi_extract, vendor_file,
  replica, analytics
  source    text not null                      // what it was made from
  contains  text not null                      // highest data class present
  readers   text not null                      // roles, logins or groups
  storage   text not null                      // where the bytes live
  owner     text not null                      // a team, never a person
  ends      timestamp null                    // null only for replicas and
  analytics
  review_due timestamp not null                // when the row is next reviewed
  deleted_by text null                        // who removed it, once removed
```

Los campos nuevos son `storage` y `deleted_by`; `kind` gana los valores `backup`, `laptop`, `bi_extract` y `vendor_file`, y `contains` registra la clase de datos más alta presente. El primer campo nuevo dice dónde mirar cuando alguien pregunta qué puede leer la copia, porque la respuesta suele ser un permiso de almacenamiento y no un rol de base de datos. El segundo cierra el ciclo: una fecha `ends` sobre la que nadie actúa es un deseo, y el registro de quién eliminó la copia es la evidencia de que la fecha significaba algo.

Backups y snapshots

Un backup es la única puerta lateral contra la que nadie discute, y ese acuerdo es parte de la razón por la que tan a menudo queda sin control. Tiene que contenerlo todo, porque su trabajo es restaurarlo todo. Ese es todo el diseño, y es también toda la exposición.

Qué contiene un backup

Los backups vienen en unos pocos tipos. Un **backup físico** o snapshot de volumen copia los archivos de la base de datos, a menudo con un archivo continuo de cambios para poder restaurar la base de datos a un punto en el tiempo. Un **dump lógico** lee los datos a través de la base de datos y los escribe como sentencias o filas. Ambos contienen todas las tablas. El tipo

físico contiene además las definiciones de roles del clúster y todo el material de credenciales que el motor almacena para ellos, y algunas herramientas lógicas también, según cómo se usen.

De ahí se siguen dos consecuencias, y ninguna es visible en un diagrama del primary. La primera es que las columnas Restricted están en el backup, y también las tablas de secrets en los motores que las usan (`bank_account_secrets` y `payment_token_values`, del Capítulo 3). La regla del esquema de roles según la cual solo los roles de la aplicación y `human_breakglass` llegan a esos valores describe la base de datos. No dice nada sobre el archivo.

La segunda es que el backup contiene el estado del día en que se tomó. Un backup anterior a la retirada del login de administrador compartido sigue conteniendo ese login. Contiene las membresías de rol de personas que desde entonces han cambiado de equipo. La rotación y la revocación ocurren en el sistema vivo, y no alcanzan hacia atrás a los archivos que ya existen. Cuando alguien restringe un rol en el primary, nada le ocurre a la restauración del año pasado: el primary mejora, y las copias conservan el estado del día en que se hicieron.

Restaurar es leer

Pondría una frase al principio del diseño de backups: el poder de restaurar es el poder de leerlo todo. Quien pueda restaurar un backup en un lugar que controla puede después iniciar sesión allí como propietario y leer todas las columnas, y no se aplica ningún gateway, grant de columna ni regla de enmascaramiento de producción, porque la lectura no ocurre en producción.

Esto cambia a quién pertenece el control. El acceso a los backups suele concederse mediante roles de infraestructura: alguien puede leer el bucket, montar el snapshot o ejecutar la herramienta de restauración. Esos roles rara vez se revisan con el mismo cuidado que `human_readonly`, aunque están por encima de él. La pregunta para una revisión de accesos no es, por tanto, solo “quién puede consultar el primary” sino “quién puede restaurar, y en qué”.

Separaría tres poderes y los daría a principals distintos:

Poder	Qué significa	Quién lo tiene
Crear	Crear backups y escribirlos en el almacén	La identidad propia del job de backup, con solo escritura en el almacén
Restaurar	Leer un backup y levantarlo en algún lugar	El equipo de plataforma, por solicitud, con una segunda persona que aprueba (four-eyes)
Eliminar	Quitar o hacer vencer backups	El proceso de retención; las personas solo mediante una solicitud revisada

El sentido de la división es que el job que escribe los backups no puede leerlos de vuelta, y nadie puede a la vez restaurar un backup y eliminar los puntos de recuperación, de modo que a una restauración no puede seguirle la eliminación de aquello desde lo que se restauró. Donde el almacenamiento admite retención write-once durante un período, la usaría por la misma razón, con un costo: no puedes eliminar un backup que hiciste por error, como uno tomado de una base de datos que estuvo brevemente algo que no debía, hasta que termina el período. Decide de antemano cómo se maneja ese caso.

Cifrado, y de quién es la clave

Los backups deberían cifrarse en reposo, y la pregunta interesante es quién guarda la clave. Si el mismo administrador puede leer el bucket y usar la clave, el cifrado protege contra un disco perdido y nada más. Si la clave la tiene un principal distinto de los administradores de almacenamiento, y restaurar requiere a ambos, entonces el equipo de almacenamiento por sí solo no puede leer los datos y quien tiene la clave por sí solo no tiene nada que leer.

Esa separación cuesta algo concreto. Restaurar ahora requiere a dos partes, y un backup cuya clave se ha perdido deja de ser un backup. Tiene que existir una vía de recuperación de claves y tiene que probarse, lo que supone un procedimiento más para que el equipo de plataforma lo mantenga. El Capítulo 12 trata el cifrado a nivel de aplicación y los límites de la gestión de claves; el punto más acotado aquí es que el cifrado del backup es un gate sobre la restauración, y un gate solo vale la pena si es distinto del gate que hay sobre el almacenamiento.

El cifrado no ayuda contra la persona a quien se le permite restaurar. Protege el archivo en tránsito y en reposo frente a personas y sistemas que no forman parte de la ruta de restauración. Lo diría con claridad en el documento de diseño, porque “los backups están cifrados” suele entenderse como “los backups están seguros”, y son afirmaciones distintas.

Una solicitud de restauración

La restauración es el momento en que el backup se convierte en una base de datos viva, así que el control pertenece ahí. El bloque de abajo muestra una solicitud de restauración con el orden de las operaciones escrito. El almacén de backups, el registro y el registro de auditoría son sistemas separados, así que ninguna transacción los cubre.

```

function restoreBackup(request):
    if request.approver is null or request.approver ==
request.requester:
        raise SelfApprovalNotAllowed
    if request.target not in isolatedTargets:
        raise TargetNotIsolated
    if request.ends is null or request.ends > now() + maxRestoreLife:
        raise RestoreNeedsEnd

    // the register comes first: a copy that exists with no record is
a side door
    registerCopy({ name: request.target, kind: "backup", source:
request.backup_id,
                    contains: "Restricted", readers: "none until access
applied",
                    owner: request.owner_team, ends: request.ends })

    try:
        auditLogAppend({ actor: request.approver, action: "restore",
                        subject: request.backup_id, target:
request.target })
        on AuditLogUnavailable:
            raise RestoreRefused

    // set before the copy exists, so a failure part-way still ends;
if this
    // call fails, the sweeper on `ends` removes the copy all the same
    scheduleDeletion(request.target, at: request.ends)
    // reachable only by the restore tooling until the last step
    startRestore(request.backup_id, into: request.target, reachable:
false)
    // every login returns with its backup-day credential, the
superuser and
    // ledgerly_access_admin included: nothing may connect before this
is done
    disableAllLogins(on: request.target)
    rotateSuperuserCredential(on: request.target)
    // the backup already holds a login with this name: drop it and
create it
    // again with a new credential, so the backup-day one never works
    replaceLogin("ledgerly_access_admin", on: request.target,
credential: "new")
    applyAccessDefinition(on: request.target)
    openTarget(request.target, readers: request.readers)

```

Si el proceso se detiene después de `registerCopy` y antes de que empiece la restauración, el registro contiene una entrada para una copia que no existe, que una comprobación semanal encuentra y elimina. Si se detiene después de que empiece la restauración, la copia existe y es inalcanzable, y su eliminación programada la quita; si ni siquiera se logró programar, el sweeper que actúa sobre `ends` en el registro la encuentra. El orden inverso

dejaría una copia que existe y no figura en la lista, y ese es el fallo que importa. El costo del diseño es la demora: una aprobación y un destino aislado antes de que nadie vea una fila. Durante un incidente esa demora es real, y por eso la ruta de restauración debería ensayarse con la aprobación acordada de antemano, como discute el Capítulo 15 para los incidentes. Cuando el objetivo es recuperar unas pocas filas y no una base de datos entera, la restauración va al mismo destino aislado, las filas se toman de él y vuelven a producción como una corrección de datos bajo las reglas del Capítulo 10.

Una copia restaurada empieza cerrada

Una base de datos restaurada trae de vuelta los roles, las membresías y el material de credenciales del día del backup. La trataría como hostil hasta que se haya vuelto a aplicar la definición de acceso. En la práctica eso significa un destino de restauración sin ruta de red desde los servicios y sin logins humanos, y un primer paso que desactive todos los logins que trajo el backup, el superusuario y `ledgerly_access_admin` incluidos, porque cada uno vuelve con la credencial que tenía el día del backup. Después se rota la credencial del superusuario, y solo entonces se aplica la definición de acceso del Capítulo 3: roles, membresías y grants tal como los describe ahora el código. La copia pasa a ser alcanzable después de eso, y no antes.

Ese paso necesita un administrador de accesos, y tiene que ser uno que pertenezca al destino. El backup ya contiene un login llamado `ledgerly_access_admin`, que volvió con la credencial del día en que se tomó el backup. La restauración lo reemplaza: el login devuelto se elimina y se crea de nuevo con el mismo nombre y una credencial nueva, hecha para la restauración y eliminada con ella. La credencial actual de producción para ese rol nunca se copia al destino, y la que hay en el backup desaparece antes de que nada pueda conectarse, porque quien tenga cualquiera de las dos puede conceder cualquier cosa dentro de los roles que administra, y un simulacro de restauración es exactamente el lugar donde las credenciales se pegan en un entorno de pruebas. Si el propio administrador de la copia restaurada se ve comprometido, el radio de impacto es esa copia.

Solo después de eso la copia restaurada recibe lectores, y los lectores son las personas de la solicitud de restauración. Una restauración para un simulacro no necesita lectores humanos en absoluto: el simulacro comprueba que los datos levantan y que los conteos coinciden, y luego se elimina la copia.

Retención y eliminación

La retención es un equilibrio entre dos costos. Si conservas los backups más tiempo, puedes recuperarte de problemas que se notaron tarde, al precio de un montón mayor de datos que lo contienen todo, incluidos datos de clientes que el primary eliminó hace mucho. Si los conservas menos tiempo, guardas menos, y alguna corrupción será más antigua que tu backup más antiguo. No sé cuál es el período correcto para tu empresa. Depende de cuánto suele tardar en notarse un problema silencioso, de cuánto costaría un mes de datos perdido y de lo que digan tus equipos legal y de cumplimiento sobre los datos eliminados que siguen vivos en los backups. Los regímenes de privacidad como GDPR o KVKK suelen preguntar qué ocurre con los datos de una persona una vez eliminados del sistema vivo, y la interpretación es suya, no de este libro. Lo que la ingeniería puede aportar es evidencia, no cumplimiento: un registro y un registro de eliminaciones que muestren lo que hace el sistema.

Un diseño que vale la pena discutir con ellos: los backups vencen según un calendario, y un procedimiento de restauración vuelve a aplicar las eliminaciones registradas desde que se tomó el backup, de modo que una copia restaurada no trae de vuelta datos que se quitaron a propósito. Eso añade un paso a cada restauración y exige que las eliminaciones se registren en algún lugar que una restauración pueda consultar, lo cual es otra razón para mantener las solicitudes de eliminación en un registro duradero. El paso se ejecuta después de aplicar la definición de acceso y antes de que se abra la copia.

```
function reapplyDeletions(target, backupTakenAt):
    // by execution time: a deletion recorded before the backup but
    run after
    // it is not in the backup's state, so it must be applied again
    for each request in deletionRecord.executedSince(backupTakenAt):
        auditLogAppend({ actor: "restore-tooling", action:
"reapply_deletion",
                                subject: request.id, target: target }) //
record first
    deleteRows(on: target, subject: request.subject_id)
    // a failure here leaves the copy closed: openTarget has not run
yet
```

Los simulacros crean copias

Un backup que nunca se ha restaurado no ha demostrado que funcione. Los simulacros de restauración son la manera de saber que los backups funcionan, y cada simulacro crea una base de datos que contiene datos Restricted. Ejecutaría los simulacros por la misma ruta de solicitud que una restauración real, con el mismo destino aislado, la misma entrada en el registro y una fecha de fin dentro del día. El simulacro es también el ensayo del incidente: si necesita un atajo para terminar, el atajo se tomará a las tres de la madrugada.

Advertencia: Un backup o un dump que contiene columnas Restricted y está en un almacenamiento legible por un grupo amplio es la misma exposición que un grant permanente sobre esas columnas, sin gateway y sin log. Si encuentras uno, restringe primero el permiso de almacenamiento y habla de la retención después.

Dumps, exports y archivos CSV

Los backups son la puerta que tiene un propósito y un equipo. Los dumps y los exports son las puertas que suele abrir una sola persona bajo presión de tiempo, y vienen en dos tipos que deberían tratarse de forma distinta: los exports que ocurren de forma programada y los que hace una persona.

Dumps hechos a mano

La carpeta compartida de Ledgerly contiene unos cuantos dumps antiguos. Cada uno se hizo porque alguien necesitaba una copia durante una hora, y la copia se quedó. El dump de una persona es un backup sin retención, sin política de cifrado y sin responsable. Es también, en los motores donde un dump lee a través de la base de datos, evidencia de que alguien tuvo un login capaz de leer todas las tablas.

Haría tres cosas. Primera, sacar el dump del conjunto de herramientas normales: el access gateway no ofrece un dump de la base de datos entera, los roles humanos no pueden leer las tablas que un dump necesitaría, y `human_breakglass` es el único rol humano que podría, con el conjunto de logging de break-glass del Capítulo 8. Segunda, encontrar los dumps existentes y, antes de eliminar ninguno, comprobar si alguno es el único backup de alguien. Eso ocurre: un dump se convierte en el backup de una tabla que nadie más copió. Construye primero los backups reales y luego elimina los dumps. Tercera, dar a la persona que necesitaba el dump una alternativa que cumpla la función, que suele ser la copia de analítica enmascarada o un conjunto de datos de prueba (véase la sección sobre datos de prueba más abajo).

Exports humanos

Los exports que hacen personas son los más comunes y los más difíciles de prohibir. Soporte necesita una lista de las facturas de un cliente. Finanzas necesita un archivo con los pagos del mes pasado. El equipo de datos necesita una muestra. Los clasificaría por niveles según la clase de datos más alta del archivo, porque una única regla que sirva para todos es demasiado laxa para los peligrosos o demasiado pesada para los inofensivos.

Nivel	Contenido	Control
Solo Internal	Agregados, conteos, desgloses por estado, sin contenido personal	Autoservicio, registrado, sin aprobación
Contiene Confidencial	Filas con contacto de clientes o texto libre	Ticket con motivo, un aprobador, una fecha de fin para el archivo; las filas salen de una sesión <code>SELECT JIT</code> que nombra las columnas (Capítulo 8), de modo que el aprobador lee la lista de columnas que tendrá el archivo

Nivel	Contenido	Control
Contiene Restric- ted	Valores de cuentas bancarias o tokens	No se exporta; la respuesta es una forma de visualización o una ruta de la aplicación (Capítulo 12)

El nivel lo calcula el gateway, no lo declara quien solicita, y falla cerrado: es la clase de catálogo más alta (Capítulo 12) entre las columnas de origen que lee la consulta, no entre los nombres de las columnas del resultado, porque un alias o una expresión como `lower(contact_email) AS x` oculta el origen. Una columna cuyo origen no se puede resolver, a través de una vista, una CTE o una función, cuenta como el nivel exportable más alto, y donde ni siquiera eso se puede decidir, el export se rechaza. El Capítulo 9 trata los límites del análisis estático. Estos niveles son una ilustración de una forma. Lo que importa es que la aprobación va unida al riesgo. Una aprobación que se concede en segundos, siempre, para todo, es el teatro que describe el Capítulo 8. El nivel que necesita un aprobador humano debería ser lo bastante pequeño como para que el aprobador lea la solicitud, y lo mediría por si alguien llega a decir que no.

La fecha de fin es la parte que los equipos se saltan. Un export sin fecha de fin es una copia que sobrevive a su pregunta. El ticket que aprueba el archivo lleva una fecha, el archivo va a un lugar con responsable, y un sweeper lo elimina. La regla del Capítulo 10 para el trabajo ad hoc cubre el caso más ligero: pega identificadores y conteos, no filas con contenido personal. El Capítulo 7 cubre lo que el gateway puede hacer con un conjunto de resultados masivo, que es limitarlo y hacerlo visible, no impedirlo.

El export para el proveedor

El proveedor de contabilidad externo nunca se conecta a una base de datos de Ledgerly. Recibe archivos, lo que hace del paso de export una puerta que puedes diseñar por completo, porque controlas ambos extremos de la ruta hasta la bandeja de entrada del proveedor. En el estado inicial, el paso de export de `reporting-job` escribe esos archivos con el login de administrador compartido, de modo que el archivo puede contener cualquier cosa que el rol de administrador pueda leer. En el estado objetivo usa

`app_vendor_export`, y eso cambia lo que el archivo puede contener, porque un archivo no puede contener más de lo que podía leer el login que lo escribió.

```
// illustrative column lists: the vendor's real needs decide them
GRANT SELECT (id, customer_id, total_minor, currency, status, due_on,
created_at)
    ON invoices TO app_vendor_export;
GRANT SELECT (id, amount_minor, currency, account_code, created_at)
    ON ledger_entries TO app_vendor_export;
```

El grant es el control, no la consulta. Un error en la consulta del export puede dejar fuera filas o repetirlas, y no puede añadir `contact_email`, porque el rol no puede leerlo. `customer_id` es un identificador opaco y el proveedor no tiene ninguna tabla con la que unirlo. Si el trabajo del proveedor necesita nombres de clientes, esa es una decisión que alguien toma y aprueba, y no llega porque se haya ampliado una consulta.

El paso de export se conecta a `ledgerly_replica_2`, como dice la política de conexiones del Capítulo 6, y escribe un archivo. Después el archivo pasa al proveedor por el canal acordado con él. Los pasos involucran tres sistemas separados, así que el bloque de abajo muestra el orden y lo que deja cada fallo.

```

function runVendorExport(period):
    file = writeExportFile(period, as: "app_vendor_export") // local,
not yet shared
    manifest = { file: file.name, rows: file.rowCount, checksum:
file.checksum,
                columns: file.columnNames, period: period }

    // the record exists before the file leaves: a sent file with no
    // record is the failure to avoid
    registerCopy({ name: file.name, kind: "vendor_file", source:
"ledgerly_replica_2",
                contains: "Internal", readers: "accounting vendor",
                owner: "platform", ends: now() + fileLife })

    try:
        sendToVendor(file, manifest)
    on SendFailed:
        // nothing has left; the file stays local and is removed with
its end date
        notify("platform", "vendor export not delivered: " +
file.name)
        raise ExportNotDelivered

    // the file has already left: a log outage delays the entry, never
the export
    auditOrQueue(manifest)

```

Aquí `auditOrQueue` añade el registro al registro de auditoría y, si este es inalcanzable, lo pone en cola para un retry y notifica al equipo de plataforma, como en el Capítulo 7. Un fallo del log se convierte en un hueco visible y con retry, y no en uno silencioso, y ninguna transacción abarca el log y el canal del proveedor.

El manifiesto es la parte que la gente omite. Permite a ambas partes comprobar que el archivo es el que se pretendía, y da a una revisión de accesos algo que leer: tantas filas, estas columnas, este período. Si una columna aparece en un manifiesto y no estaba en el del mes pasado, es un cambio que alguien debería haber aprobado.

Quedan dos límites. Una vez que el archivo ha llegado al proveedor, su retención es una cuestión de contrato y de confianza: puedes pedirle al proveedor que confirme la eliminación, y no puedes comprobarla. Y el checksum muestra que el archivo no se alteró en tránsito, no que el personal del proveedor sea la única gente que lo lee. Para esos dos pondría un término de retención en el acuerdo y una línea en la revisión de accesos que pregunte cuándo confirmó el proveedor por última vez la eliminación.

Datos de prueba: staging, copias enmascaradas y datos sintéticos

Staging es donde empezó el escenario de Ledgerly, y es la puerta que se abre más a menudo por amabilidad. Los ingenieros quieren datos realistas, y producción es la única fuente de datos que tiene las formas que crean los clientes reales. El costo es que staging tiene el modelo de acceso de un sistema de pruebas, con los datos de uno de producción.

Tres tipos de datos que no son de producción

Tipo	Realismo	Exposición	Costo
Copia de producción	El más alto	La misma que producción, con controles más débiles	Barata de hacer, cara de gestionar
Copia enmascarada	Alto, con la pérdida de algunas formas	Menor, no cero	Un pipeline de enmascaramiento que construir y mantener (Capítulo 12)
Datos sintéticos	Menor, limitado a lo que se modeló	La más baja	Un generador que mantener, y bugs que necesitan formas no modeladas

No copiaría datos de producción a staging sin enmascarar, y la razón no es abstracta. Staging suele tener más lectores que producción, un login compartido, CI que guarda logs e integraciones configuradas para comportarse como en producción. Combina la mayor audiencia con los controles de identidad más débiles, y contiene los datos más sensibles. Si los datos tienen que ser reales, los controles tienen que ser tan fuertes como los de producción, y en ese punto staging es producción con otro nombre.

Lo que sigue siendo una copia enmascarada

El enmascaramiento, tratado en el Capítulo 12, reemplaza u oculta valores manteniendo la forma utilizable. Una copia enmascarada es mucho mejor que una sin tratar, y sigue siendo una copia de datos, por razones que conviene enunciar para que nadie trate “enmascarada” como sinónimo de “segura para repartir”.

Primero, el enmascaramiento cubre las columnas que listaste. El texto libre como `invoice_lines.description` puede contener datos personales que la clasificación nunca anticipó, razón por la cual el libro lo clasifica

pensando en el peor caso, y una máscara que lo omite lo deja en la copia. Segundo, las máscaras que preservan la forma conservan filas que siguen siendo distinguibles: un importe de factura inusual, una fecha inusual y un cliente pequeño pueden juntos apuntar a un solo negocio. Los valores estaban enmascarados, y la combinación sigue llevando información. Tercero, el enmascaramiento consistente, donde la misma entrada da la misma salida para que los joins sigan funcionando, también permite reconocer a la misma persona entre copias y tablas.

Nada de esto significa que no valga la pena enmascarar. Significa que una copia enmascarada se trata como Confidential hasta que alguien haya revisado lo que queda, su audiencia es más pequeña que “toda la ingeniería” y tiene una fecha de fin como cualquier otra copia; la copia de analítica continua es la excepción, y se revisa en su fecha `review_due`. La palabra para esto es **reidentificación**: la capacidad de deducir de quién trata un registro a partir de los campos que se dejaron. Describiría el riesgo al equipo en esos términos y no en el lenguaje de los ataques. La respuesta de diseño es una audiencia más pequeña y un vencimiento, no una lista más larga de campos enmascarados.

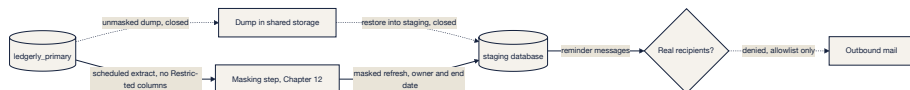
Datos sintéticos

Los datos sintéticos no contienen a ninguna persona real, así que llevan la menor exposición. Su límite es que solo contienen las formas que alguien pensó en modelar, y los bugs que necesitan datos reales son los que nadie modeló. Un generador que hace clientes ordenados no hace al cliente con seis contactos y una descripción en dos idiomas.

Haría crecer el conjunto sintético a partir de los bugs. Cada vez que un problema necesita datos con forma de producción para reproducirse, la forma que lo causó, y no los valores, se convierte en un fixture: un cliente con varios contactos, una línea de factura con una descripción vacía, una fecha de vencimiento a fin de mes. En un año el conjunto deja de ser ordenado y empieza a ser el tipo útil de raro. Cuesta a un ingeniero una hora después de cada bug así, y elimina la necesidad de la siguiente copia de producción.

El staging de Ledgerly, rehecho

Volvamos al viernes de Robin. El camino que abrió la puerta fue un dump del almacenamiento compartido restaurado en staging. El camino que construiría lo reemplaza con un refresh que pasa por el paso de enmascaramiento y aterriza en una base de datos con vida corta y un responsable con nombre. En el diagrama, mira las rutas discontinuas: son las que el diseño cierra.



La mitad derecha del diseño es fácil de olvidar. Un entorno de staging contiene datos y además actúa sobre ellos. `invoice-worker` en staging lee clientes y envía recordatorios, y si sus ajustes de salida coinciden con los de producción, los envía a las direcciones de contacto que hay en los datos. Una copia enmascarada no tiene direcciones reales, lo que lo arregla por casualidad. Una copia sin tratar convierte una ejecución de prueba en mensajes a clientes reales. Haría que la ruta de salida de todo entorno que no sea de producción niegue por defecto y permita solo una allowlist de direcciones de prueba internas, sea cual sea el contenido de los datos.

Los tokens de pago siguen una regla relacionada. Un token representa un instrumento de pago, y que funcione fuera del sistema que lo emitió depende del proveedor, algo que no puedo juzgar para el tuyo. Por defecto, `payment_tokens.token_value`, `bank_accounts.iban` y `bank_accounts.account_holder_name` nunca están presentes en una copia que no sea de producción, enmascarada o no (los backups son la excepción, porque deben contenerlo todo), y staging usa los tokens de prueba del proveedor.

Nota: Trata el refresh de staging como un pipeline con una ruta de fallo, no como algo puntual. Si falla el paso de enmascaramiento, el refresh se detiene y staging conserva su contenido anterior. Un refresh que recurre a una fuente sin enmascarar por una vez recrea el dump de la carpeta compartida con pasos adicionales.

Herramientas de BI, laptops y proveedores

Tres puertas más merecen un tratamiento breve, porque cada una saca datos de la base de datos por una vía que parece una funcionalidad.

Herramientas de BI. Una herramienta de dashboards se conecta con un login, y todos los que tienen acceso al dashboard ven lo que ese login puede leer. Eso invierte el modelo de identidad del Capítulo 3: la base de datos ve un principal y, digamos, cien personas ven sus datos. Dos controles ayudan. Apunta la herramienta a `ledgerly_analytics` y no a una réplica, de modo que lea datos enmascarados con un rol propio para la herramienta, un rol `app_` como cualquier otra carga de trabajo (`human_analytics_reader` es el rol del equipo de datos para personas), en lugar de roles de producción. Y, donde la herramienta lo admita, haz que pase la identidad de cada persona a la base de datos, de modo que las reglas de fila y de columna se apliquen por persona. La herramienta también guarda extractos, caches y informes programados por email propios, que son copias de datos y van en el registro. Un dashboard es una puerta cómoda, y un extracto en cache es la misma puerta sin los controles.

Laptops. El gateway elimina las credenciales guardadas y establece valores por defecto de solo lectura, como describe el Capítulo 7, y no puede controlar lo que una persona hace con las filas que llegaron a la pantalla. El control práctico es reducir los motivos para conservar una copia local: una muestra enmascarada, una base de datos de staging que se comporte como producción y fixtures para las formas que causan bugs. Un equipo no puede demostrar que ningún laptop contiene datos de producción, y yo no lo afirmaré. Una línea base como el cifrado de disco completo y una manera de borrar un dispositivo perdido reduce lo que cuesta perder un laptop, y es una línea base, no una respuesta.

Otros proveedores. La contabilidad es el que sigue el libro, y otros reciben datos por efecto secundario: un rastreador de errores que captura el cuerpo de una solicitud, un servicio de atención que recibe capturas de pantalla pegadas, una plataforma de logs que almacena sentencias. Cada proveedor que tiene una copia recibe una fila en el registro con lo que contiene y cuándo termina el contrato. El Capítulo 12 trata cómo mantener los valores sensibles fuera de logs y errores, que es la manera más barata de mantenerlos fuera del almacén de un proveedor.

Cerrar las puertas en Ledgerly

Considera cómo llevaría Alex a Ledgerly de copias dispersas a un registro con responsables. Como con la ruta de lectura del Capítulo 6, el orden importa, y lo haría en seis pasos.

1. **Inventariar las copias.** Lista backups, snapshots, dumps, exports, staging, extractos de BI y archivos para proveedores, con almacenamiento, lectores y un responsable. Espera encontrar copias que nadie recuerda; la primera ejecución del registro es sobre todo descubrimiento.
2. **Restringir primero el almacenamiento.** Para todo lo que contenga columnas Restricted, acota quién puede leer el almacenamiento antes de decidir qué eliminar. Es rápido y reversible, y cierra la exposición más amplia mientras se ordena el resto.
3. **Construir los poderes de backup.** Da al job de backup una identidad con solo escritura, pon la restauración detrás de una solicitud con una segunda persona que aprueba, guarda la clave de cifrado aparte de los administradores de almacenamiento y ejecuta un simulacro por la ruta de solicitud.
4. **Retirar los dumps.** Una vez que existan los backups reales y haya funcionado una restauración, elimina los dumps antiguos y registra quién eliminó cada uno. La restauración nocturna sin enmascarar que alimenta `ledgerly_analytics` termina cuando la carga enmascarada del Capítulo 12, bajo `app_analytics_load`, se ejecuta en su lugar; hasta entonces es la mayor puerta abierta y recibe una fecha de fin.
5. **Pasar staging a datos enmascarados o sintéticos** con los mensajes salientes desactivados por defecto, y eliminar el login compartido de staging.
6. **Poner el export para el proveedor bajo su propio login.** Pasa el paso de export a `app_vendor_export`, añade el manifiesto y fija una vida para el archivo.

Después conecta el registro a algo que ocurre de todas formas. La revisión de accesos del Capítulo 14 pregunta qué copias existen y quién puede leer cada una, crear una copia exige una fila en el registro, y un sweeper elimina todo lo que pase de su `ends` tras un período de gracia y una notificación al responsable. Las solicitudes de restauración y los exports también

deberían escribir en el registro de auditoría, que es el tema del Capítulo 14. Los jobs de restauración y de export siguen la misma disciplina de ejecución que el runner que lleva las escrituras aprobadas: se ejecutan bajo su propia identidad, a partir de una solicitud aprobada, con el registro escrito primero, salvo que un export del nivel Internal conserva la regla de autoservicio de la tabla de niveles de arriba (el Capítulo 9 describe esa disciplina).

Trade-offs y costos

El primer costo es la velocidad durante un incidente. Una restauración que necesita un aprobador y un destino aislado es más lenta que una hecha por quien tenga el permiso del bucket, y el tiempo que añade recae sobre quien esté de on-call en el peor momento. Ensaya la ruta y acuerda la aprobación de antemano (el Capítulo 15 trata el acceso durante los incidentes), de modo que el control sea un procedimiento que la gente conoce y no un obstáculo que descubre.

El segundo costo es la custodia de la clave. Separar la clave del backup del almacenamiento significa que una clave perdida es un backup perdido, y la vía de recuperación de la clave es un procedimiento más que hay que escribir y probar. Para un equipo muy pequeño la separación puede costar más de lo que devuelve; un almacén cifrado, permisos de almacenamiento ajustados y una recuperación de claves documentada pueden ser un punto de partida razonable.

El tercer costo es un ciclo de depuración más lento. Cuando los ingenieros no pueden restaurar producción en staging, algunos bugs tardan más en reproducirse, porque la forma que los desencadena todavía no está en los datos de prueba. El hábito de los fixtures lo compensa con el tiempo y es un gasto real al principio. Las copias enmascaradas añaden un pipeline del que alguien es responsable, y se romperá cuando se añada una columna, razón por la cual la migración que añade una columna es donde se decide su clase (Capítulo 11).

El cuarto costo es la retención. Una retención más larga mantiene más historial recuperable y mantiene más datos sensibles en más lugares; una retención más corta invierte ambas cosas. No hay un valor que sea correcto para todos, y un equipo que no ha elegido uno ha elegido “para siempre”.

Por último, el registro y su sweeper son sobrecarga. Un sweeper que elimina una copia que alguien todavía necesitaba es un incidente en sí mismo, así que la eliminación debería llegar tras una notificación y un período de gracia, y un responsable debería poder prorrogar una fecha de fin una vez con un motivo. Un registro que nunca se abre no protege nada, y el vínculo con la revisión de accesos es lo que lo mantiene honesto.

Cuándo no construir todo esto: una empresa pequeña con una base de datos, un puñado de ingenieros y sin datos Confidential dignos de ese nombre no necesita aprobación de exports por niveles ni una clave dividida. Aun así necesita el inventario, almacenamiento restringido para los backups y una regla de que nada de producción aterriza en staging sin enmascarar. Haz esas tres cosas, deja por escrito qué te haría hacer más y revisa esa nota cuando cambien el equipo o los datos.

Errores comunes

- **Restringir el acceso al primary e ignorar sus backups.** Hay roles, un gateway y un registro de auditoría, y el bucket de backups es legible por todo el grupo de infraestructura. Es tentador porque los backups son “operaciones”. El remedio es tratar la restauración como una lectura de todo, con su propio aprobador y su propio log.
- **Creer que “cifrado” significa “seguro”.** El cifrado en reposo protege el archivo de quienes están fuera de la ruta de restauración, y la clave está junto al bucket en manos del mismo administrador. Guarda la clave aparte y di contra qué no protege.
- **Eliminar los dumps antiguos antes de que existan backups reales.** El dump era la única copia de alguien de una tabla, y quitarlo quitó la única vía de vuelta. Construye y prueba primero los backups, y luego retira los dumps.
- **Restaurar en una red donde los logins antiguos funcionan.** La copia restaurada trae los roles y las credenciales de su día, incluidos los retirados desde entonces. Restaura en aislamiento, vuelve a aplicar la definición de acceso y reemplaza el login `ledgerly_access_admin` que trajo el backup por uno nuevo en lugar de conservar cualquiera de las credenciales antiguas.

- **Tratar una copia enmascarada como pública.** Los valores están enmascarados y la copia se comparte ampliamente; luego el texto libre o una combinación rara de campos apunta a un cliente. Trátala como Confidential, con una audiencia más pequeña y una fecha de fin, hasta que se revise.
- **Refrescar staging desde producción como una excepción puntual.** El paso de enmascaramiento falló, el release esperaba y el dump estaba ahí mismo. Haz que el refresh falle cerrado, de modo que los datos antiguos se queden y nada recurra a una fuente sin tratar.
- **Dejar que una herramienta de BI se conecte con un rol de producción.** Un login detrás de, digamos, cien espectadores, con un cache que nadie listó. Conéctala a la copia de analítica con su propio rol y registra sus extractos.
- **Aprobar todos los exports en segundos.** La aprobación existe y no reduce nada. Reserva al aprobador para el nivel que merece una decisión, y observa si alguien llega a rechazar.
- **Dar a una copia un responsable y ningún final.** El registro está completo y nada sale de él. Cada fila necesita una fecha `ends` y un sweeper que actúe sobre ella, con una notificación previa.

Ideas clave

- Una puerta lateral lleva los datos sin los controles del sistema del que vinieron, porque los controles de acceso viven en la base de datos y no en el archivo.
- El poder de restaurar un backup es el poder de leer todo lo que contiene, así que el acceso a los backups merece la misma revisión que el rol humano más privilegiado.
- Crear, restaurar y eliminar backups debería recaer en principals distintos, y la clave de cifrado debería guardarse aparte de los administradores de almacenamiento, con una vía de recuperación probada.
- Una copia restaurada vuelve con los roles y las credenciales del día en que se tomó, así que empieza aislada, se le vuelve a aplicar la definición de acceso y recibe su propio login de administrador.
- Un export para un proveedor se controla con el grant de `app_vendor_export`, no con la consulta, de modo que el archivo no

puede contener una columna que el rol no puede leer, y un manifiesto permite comprobar el archivo.

- Los datos de prueba deberían ser enmascarados o sintéticos, una copia enmascarada se trata como Confidential hasta que se revise, y staging debería denegar por defecto los mensajes salientes y fallar cerrado cuando falla el enmascaramiento.
- Toda copia necesita un responsable y una fecha de fin en un registro que se revisa junto con los accesos, porque una puerta que nadie posee es una que nadie cierra.

Las copias, los exports y las restauraciones ya tienen responsable y fecha de fin, pero un diseño solo vale lo que valga la evidencia de que se cumple. El siguiente capítulo se ocupa de la auditoría: dónde vive el registro de auditoría, qué entra en él y cómo se ejecutan realmente las revisiones de accesos que este libro menciona una y otra vez.

PARTE V

Demostrar y operar

Capítulo 14 — Auditoría y rendición de cuentas

Imagina Ledgerly un martes por la mañana. Un cliente ha preguntado quién, dentro de la empresa, miró su cuenta de pagos el mes pasado, y el contacto de seguridad le reenvía la pregunta a Alex. Suena a una búsqueda. Alex empieza por la lista de sesiones del access gateway, que muestra que Robin tuvo una sesión break-glass en el primary durante cuarenta minutos. Luego Alex abre el log de la base de datos, que nombra un login de sesión pero no el motivo de la sesión. Después busca en `audit_events`, que registra las acciones de negocio que realizó la aplicación y no dice nada de lo que escribió una persona.

Las tres fuentes coinciden en la mayor parte de la historia y discrepan en el resto. El reloj del gateway va dos minutos adelantado respecto al de la base de datos. Un login de sesión aparece en el log de la base de datos sin ninguna entrada en el gateway, y nadie puede decir si es un login de servicio que la lista olvidó o una conexión que llegó por una ruta que nadie diseñó. Alex puede decir qué ocurrió probablemente, pero no puede demostrarlo, ni demostrar que nadie con derechos de administrador de la base de datos editó la evidencia.

Este capítulo atiende el pilar de rendición de cuentas de la tesis y añade el último paso del estado objetivo antes de los incidentes: el acceso se registra en un almacén que el sistema al que se accede no puede reescribir, y se revisa con una periodicidad fija. En el modelo de cinco partes (**Identidad, Grant, Ruta, Ejecución, Registro**), este es el capítulo sobre el Registro. La Identidad dice quién actuó; el Registro es lo que queda, y si vincula la sentencia con la persona, la solicitud y el resultado. La pregunta aquí es esta: ¿qué debería dejar tras de sí una sentencia, dónde debería vivir y cómo te enteras de que la evidencia está incompleta antes de que alguien pregunte?

El problema y por qué es difícil

En el estado inicial la pregunta no tiene respuesta. El log de la base de datos nombra un rol compartido y ninguna persona, y está en hosts que controlan los administradores. Los capítulos anteriores arreglaron la mitad de la identidad: un login por sesión, una solicitud por grant, un runner para las escrituras aprobadas. Cada uno de ellos añadía al registro mediante la misma llamada, `auditLogAppend`, sin decir adónde van las entradas. Este capítulo lo responde.

La dificultad es que Ledgerly ya tiene tres registros, y cada uno sabe algo distinto. Ninguno es el registro de auditoría tal como lo define el libro: el registro de quién hizo qué sobre qué datos, guardado donde quien actúa no puede modificarlo.

Registro	Qué sabe	Qué no puede decirte	Quién puede modificarlo
Registro de sesiones del gateway	La persona, la solicitud, el motivo, la sentencia tal como se envió	Qué hizo realmente la base de datos; si existía otra ruta	El equipo de plataforma, a través del almacén propio del gateway
Log de la base de datos	El login, la hora, el texto de la sentencia si se registra, los errores	La persona detrás del login, la solicitud, el motivo	Quien administre el host de la base de datos
<code>audit_events</code>	La acción de negocio, el principal que actúa, el sujeto	Las sentencias que la produjeron; cualquier cosa que la aplicación no escribió	Nadie mediante <code>UPDATE</code> , pero <code>ledgerly_owner</code> puede alterar la tabla en una migración

La respuesta obvia es subir el nivel de logging: registrar cada sentencia en la base de datos y conservarla mucho tiempo. Eso añade un cuarto registro y no cierra ninguna de las tres brechas. Un log que el sistema al que se accede escribe sobre sí mismo es tan confiable como sus administradores, y un log que nadie lee es teatro de seguridad. Un log de sentencias es en sí mismo una copia de datos sensibles, porque las personas escriben literales en las condiciones; el Capítulo 12 lo advirtió, y este capítulo se ocupa del mecanismo.

El registro existe para encontrar la brecha en el sistema, no para clasificar a las personas que pasaron por él. Cuando salta una alerta sobre la sesión de Robin, la primera pregunta es qué hizo necesario el diseño, no qué hizo mal Robin.

Un registro por sentencia

Un registro solo es útil si puede responder a una lista de preguntas escritas de antemano. Escribiría primero la lista y derivaría de ella los campos, porque un almacén de auditoría diseñado a partir de las líneas de log que el gateway casualmente produce responde las preguntas que esas líneas casualmente permiten.

El gateway escribe dos entradas por sentencia, porque el Capítulo 9 exige que la intención se escriba antes de ejecutar la sentencia y el resultado después. Juntas forman el **registro de sentencia**; las dos entradas comparten un `statement_id`, y la segunda rellena los campos que se desconocían al principio. Una sola forma sirve para ambas y para los registros que escriben otros principals.

```

table access_audit_record:
  stream      text not null    // one per producer instance
  sequence    integer not null // gapless within a stream
  previous_hash text not null  // hash of the previous entry
  entry_hash   text not null
  kind         text not null    // "intent", "outcome", "event"
  action       text null        // events only: grant, revoke,
breakglass, data-fix, contain, and so on
  statement_id text null
  session_id   text null
  login        text not null    // what the database log shows
  principal    text not null    // the person or workload
identity
  on_behalf_of text null        // the human behind an
automation principal
  roles        text not null    // role set at the time of the
statement
  request_id    text null        // always access_request.id; a
data fix has one too, and its fix_id resolves through it
  reason       text null
  approver      text null
  statement_class text null      // the Chapter 9 class the
classifier assigned
  fingerprint   text null        // statement shape, literals
redacted
  literals_hash text null        // keyed hash of the literal
values
  tables        text null        // tables touched
  columns       text null        // columns touched, by data
class
  rows_affected integer null     // outcome only
  rows_returned  integer null     // outcome only
  outcome        text null        // ok, refused, error, rolled-
back, row-cap-closed
  started_at     timestamp not null
  ended_at       timestamp null

```

El tipo `event` recoge todo lo que no es una sentencia: la apertura de una sesión, un grant, una revocación, una rotación, una restauración, el inicio de una migración. Son las llamadas `auditLogAppend` y `audit0rQueue` de los capítulos 3, 4, 7, 8, 10, 11, 12, 13 y 15, y la columna `action` distingue un grant de una revocación o de una contención. Usan la misma cadena y los mismos campos de identidad, de modo que una sola consulta puede seguir a una persona desde una solicitud hasta un grant, luego hasta las sentencias que lo usaron y por último hasta la revocación.

La tabla solo vale la pena si responde a las preguntas que la gente hace. La segunda lista es la que pegaría junto al esquema.

Pregunta	Campos que la responden
¿Quién ejecutó esta sentencia?	<code>principal</code> , <code>on_behalf_of</code> , <code>login</code>
¿Con qué grants y qué aprobación?	<code>roles</code> , <code>request_id</code> , <code>approver</code> , <code>reason</code>
¿Qué pedía?	<code>statement_class</code> , <code>fingerprint</code> , <code>literals_hash</code> , <code>tables</code> , <code>columns</code>
¿Qué hizo?	<code>outcome</code> , <code>rows_affected</code> , <code>rows_returned</code>
¿Tocó datos Restricted?	<code>columns</code> clasificadas por clase de datos, con <code>rows_returned</code>
¿Cuándo y durante cuánto tiempo?	<code>started_at</code> , <code>ended_at</code>
¿Falta algo a su alrededor?	<code>stream</code> , <code>sequence</code> , <code>previous_hash</code> , la intención sin resultado
¿Esta entrada está tal como se escribió?	<code>entry_hash</code> , comprobado por <code>verifyChain</code> más abajo

Dos campos cargan con la mayor parte del peso. `principal` y `on_behalf_of` evitan que el rastro termine en una máquina, como exige el Capítulo 9 para la automatización. El `request_id` es la clave de unión con todo lo que registraron los capítulos 8 y 10.

Un registro sin `request_id` sigue siendo válido. Una sesión rutinaria `human_readonly` no tiene solicitud, y su registro lo dice dejando el campo vacío. Un `request_id` vacío en una sesión que tiene tripletas `human_jit_write` o `human_breakglass` es un hallazgo, y las comprobaciones que aparecen más adelante en este capítulo lo buscan.

Un almacén que el sistema auditado no puede reescribir

El lugar donde viven las entradas importa más que su formato. El Capítulo 1 definió la observabilidad como un registro que está en un sitio que el principal no puede cambiar. Los principals que importan aquí son los que más poder tienen: los administradores de la base de datos, el equipo de plataforma que opera el gateway y `ledgerly_access_admin`, cuyos tres

logins pueden conceder casi cualquier cosa dentro de los roles que administra. Si cualquiera de ellos puede editar el registro, el registro es evidencia solo de lo que decidieron dejar.

Construiría el almacén con cinco propiedades, en este orden.

1. **Una cuenta separada.** El almacén de auditoría vive en una cuenta de nube, un proyecto o un grupo de hosts en los que los administradores de la base de datos, los operadores del gateway y los tres logins de `ledgerly_access_admin` no tienen ningún derecho. Alguien tiene que administrarlo; es un grupo pequeño que no tiene derechos de administrador de la base de datos, y su propio uso del almacén es visible para alguien ajeno a él.
2. **Un productor de solo escritura.** Cada productor (el gateway, el runner, el credential broker, el pipeline, el job de revocación, las herramientas de restauración) tiene su propia credencial, que puede llamar a una sola operación: `append`. No puede leer, actualizar ni borrar. Una credencial de productor filtrada puede añadir entradas falsas, algo que una cadena permite detectar, y no puede eliminar las verdaderas.
3. **Un servicio de `append` que mantiene la cadena.** Los productores no calculan hashes. Un pequeño servicio dentro de la cuenta de auditoría carga la cabeza del stream, enlaza la nueva entrada y guarda ambas. Un productor que calculara su propio enlace podría también recalcular todo lo que viene después.
4. **Una retención que el administrador de la cuenta no puede acortar.** Donde el almacenamiento ofrece un bloqueo de retención de escritura única, úsalo durante el período de retención, de modo que nadie en la cuenta pueda borrar una entrada antes de que termine. Donde no lo ofrece, lo honesto es decir que el administrador puede borrar, y que la cadena y el ancla de más abajo son lo que hace visible un borrado.
5. **Un ancla.** Una vez al día el servicio de `append` envía el hash de cabeza de cada stream a un segundo lugar, propiedad de un grupo distinto: el sistema de los contactos de seguridad, o un segundo almacén. Sin ancla, una persona con control total del almacén podría reescribir un stream desde cierto punto en adelante, recalculando todos los hashes posteriores. Con ella, la cabeza reescrita difiere del ancla.

```

function appendChained(stream, record):
    // runs inside the audit account; producers can call only this
    // one writer per stream at a time: the head is read and locked
    // inside the same transaction that stores the next entry
    transaction:
        head = lockHead(stream)
        entry = { stream: stream, sequence: head.sequence + 1,
                  previous_hash: head.hash, body: record,
                  stored_at: now() }
        entry.entry_hash = hashOf(entry.stream, entry.sequence,
                                   entry.previous_hash, entry.body)
        // one system, so one transaction is allowed: the entry and
the
        // new head exist together or not at all
        storeEntry(entry)
        storeHead(stream, entry.sequence, entry.entry_hash)
    // commit point: the entry is durable and the chain has moved
    return { sequence: entry.sequence, hash: entry.entry_hash }

```

Dos sesiones pueden hacer append al mismo stream en el mismo instante, así que la cabeza se lee bajo un bloqueo dentro de la transacción; sin él, ambas leerían la misma cabeza, escribirían el mismo número de secuencia y bifurcarían la cadena, y el verificador informaría entonces de una manipulación que nunca ocurrió. La regla es un solo escritor por stream a la vez, impuesta por ese bloqueo.

El verificador es un job aparte, con una credencial de solo lectura en la misma cuenta. Su trabajo es la parte que convierte un almacén en evidencia:

```

function verifyChain(stream, from, to):
    previous = loadEntryBefore(stream, from)
    for each entry in entries(stream, from, to):
        if entry.sequence != previous.sequence + 1:
            notify("security", "hole in " + stream + " at " +
entry.sequence)
        if entry.previous_hash != previous.entry_hash:
            notify("security", "broken link in " + stream)
        if entry.entry_hash != hashOf(entry.stream, entry.sequence,
                                       entry.previous_hash, entry.body):
            notify("security", "entry altered in " + stream)
        previous = entry
    // compare at the anchor's own position, not at the end of the
range
    anchor = loadAnchor(stream, atOrBefore: to)
    if anchor is not null and anchor.hash != entryAt(stream,
anchor.sequence).entry_hash:
        notify("security", "head differs from anchor in " + stream)

```

Ahora el límite honesto, que pondría en el documento de diseño como una frase aparte. Este diseño da evidencia, no prevención. No impide que se ejecute una sentencia, y no impide que alguien con suficiente poder sobre la cuenta de auditoría destruya el almacén. Hace detectable la alteración y reduce a las personas que podrían alterarlo a una lista corta y separada. La detección depende además de que alguien lea la salida de `verifyChain`, así que el verificador es un job programado con su propio heartbeat, que se trata en la sección siguiente.

El costo es real. La cuenta de auditoría es otro sistema que mantener disponible, y los capítulos 8 y 9 ya aceptaron su peor consecuencia: se rechaza un grant, una declaración de break-glass o una sentencia cuando no se puede escribir su intención. Una caída del almacén de auditoría bloquea, por tanto, el trabajo privilegiado, y lo bloquea en el momento en que alguien puede estar esperando por una emergencia. Ese es el precio de un registro que no depende de la buena voluntad.

Advertencia: No pongas el almacén de auditoría en la base de datos que describe, ni siquiera en un esquema con su propio propietario. Quien pueda administrar esa base de datos puede alterar las entradas, y un rol que solo permite `INSERT` no cambia eso. La única excepción estructural que el libro ya enumera, que `ledgerly_owner` puede alterar `ledger_entries` y `audit_events` en una migración, es la razón por la que `audit_events` es el registro de negocio de la aplicación y no el registro de auditoría.

Encontrar los huecos

Una cadena demuestra que las entradas presentes no se alteraron. No demuestra que cada sentencia produjera una entrada. La segunda familia de comprobaciones busca lo que debería existir y no existe. Hay cuatro tipos de hueco, y cada uno significa algo distinto, así que la comprobación los informa por separado.

Huecos de secuencia. Un stream no tiene huecos por construcción, así que un hueco significa que el servicio de append perdió una entrada o que alguien la quitó. Es el hallazgo más raro y el más grave.

Intenciones sin resultado. El gateway escribe una intención antes de una sentencia y un resultado después. Una intención sin resultado una vez pasado el tiempo de la sentencia más larga que permite el timeout significa que el gateway falló entre las dos, que el resultado está esperando en la cola de retry o que la sentencia sigue ejecutándose. Solo lo tercero es inofensivo, y no debería durar más que el timeout.

Sesiones en un solo lado. Todo login de la base de datos debería tener un origen conocido. Para las personas, el origen es una sesión del gateway. Para los servicios, es un lease del credential broker del Capítulo 4. Para las migraciones, es una ejecución del pipeline. Un login en el log de la base de datos que no tenga ninguno de los tres es una ruta que nadie diseñó, y una sesión del gateway que dice haber ejecutado sentencias que la base de datos nunca vio es un registro que describe algo que no ocurrió.

Productores silenciosos. Un stream que deja de escribir puede significar una semana tranquila, un productor muerto o un productor que alguien apagó. La comprobación compara cada stream con una cadencia esperada.

```
function detectGaps(window):
    gatewaySessions = sessionsFromAuditStore(window)
    databaseLogins = loginsInDatabaseLog(window)
    for each login in databaseLogins:
        if login not in gatewaySessions
            and not hasBrokerLease(login, window)
            and not isPipelineRun(login, window):
            notify("security", "login with no known origin: " + login)
    for each s in gatewaySessions:
        if s.statements > 0 and s.login not in databaseLogins:
            notify("platform", "gateway session with no database
login: " + s.id)
        for each i in intentsWithoutOutcome(olderThan: maxStatementAge):
            if not inRetryQueue(i):
                notify("platform", "intent with no outcome: " +
i.statement_id)
        for each stream in expectedStreams():
            if now() - lastEntry(stream) > expectedCadence(stream):
                notify("platform", "stream silent: " + stream)
```

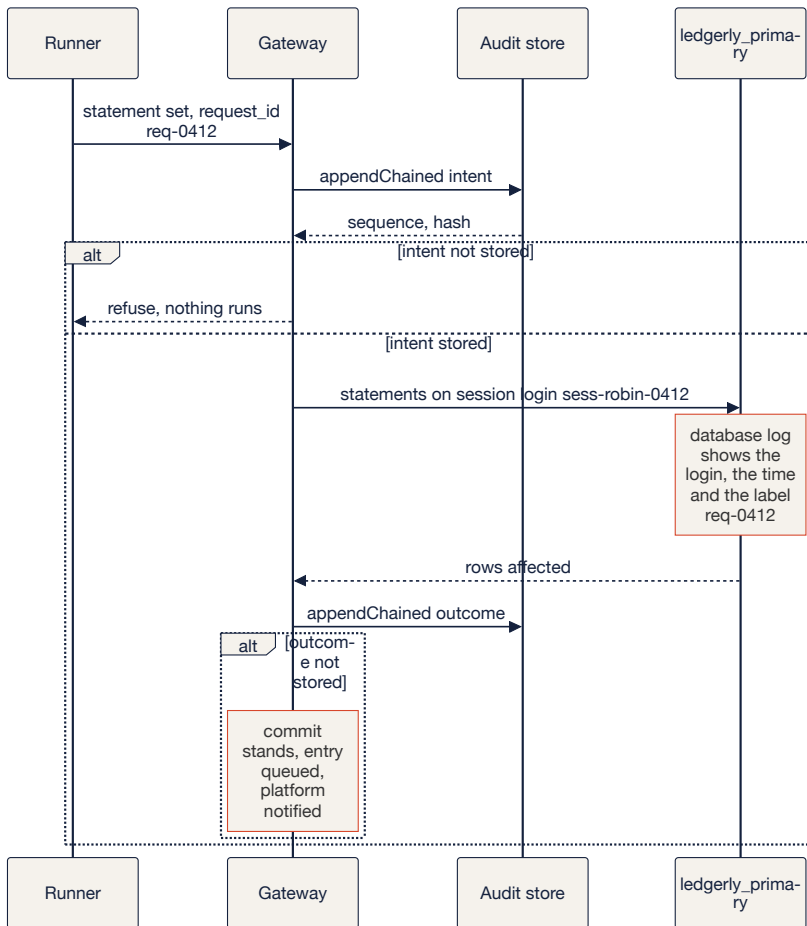
El log de la base de datos llega a la cuenta de auditoría mediante un agente de envío con su propia credencial de solo escritura. Alguien con acceso al host puede detener ese agente, y para eso existe la comprobación de stream silencioso: un agente detenido deja un stream sin entradas, y eso genera su propio hallazgo.

Unir tres registros

El registro del gateway, el log de la base de datos y `audit_events` los producen tres sistemas que nunca comparten una transacción. Unirlos es lo que convierte los casos de la mañana de Alex de conjeturas en una línea de tiempo, y las claves de unión hay que elegirlas antes del incidente, porque después ya no se pueden añadir.

Tres claves hacen el trabajo. El **login** aparece en el log de la base de datos y, como un login de sesión corresponde a exactamente una persona o una instancia de workload, vincula una línea del log con un registro de sesión. El **request id** lo escribe el gateway en el registro de auditoría y el runner en el campo `request_id` del propio almacén de auditoría; vincula la sentencia con la aprobación que la permitió. El runner nunca escribe en `audit_events`, algo que la matriz del Capítulo 9 le deniega. Las filas de `audit_events` que llevan el correlation id que introdujo el Capítulo 3 proceden de la ruta de producto, donde `admin-console` llama a `payments-service`, y se unen a la misma solicitud mediante ese id. La **etiqueta de conexión** del Capítulo 5 puede llevar el request id hasta el log de la base de datos en las conexiones mediadas. Esa etiqueta la fija quien abre la conexión. En la ejecución mediada la abre el gateway con el request id de la solicitud aprobada, así que es confiable; en un session handout, el cliente de la persona podría fijarla a cualquier valor, así que el login sigue siendo la clave y la etiqueta sigue siendo una pista.

El diagrama muestra una corrección de datos aprobada y dónde se escribe cada registro; el runner llega al primary solo a través del gateway. Si falla la escritura del resultado, el commit se mantiene y el hueco es detectable.



Lo que el registro no debe contener y quién audita a los auditores

Un registro de sentencia contiene lo que la gente escribió. Una persona que busca a un cliente por número de cuenta ha puesto el número de cuenta en la sentencia, y un log que guarda la sentencia ha copiado un valor Restricted en un lugar sin protección a nivel de columna. La advertencia del Capítulo 12 se aplica por completo, así que quien construye el registro tiene que decidir qué se conserva.

Guardaría la sentencia en dos partes. El **fingerprint** es la sentencia con cada literal sustituido por un marcador, de modo que `UPDATE invoices SET status = ? WHERE id IN (?)` registra la forma y nada más. El **hash de literales** es un hash con clave de los valores literales, que permite a

quien investiga contrastar un valor sospechoso con el registro sin que el registro lo contenga. La clave importa: literales como un estado o un sufijo corto de cuenta tienen poca entropía, y un hash sin clave podría revertirse probando candidatos. La clave vive con la cuenta de auditoría, separada de los productores, y la herramienta de quien investiga pide una comparación y nunca la clave.

```
function buildStatementRecord(session, statement, analysis):
    // analysis comes from the gateway's query analysis, not from the
    client
    parsed = parse(statement)
    record = { kind: "intent", session_id: session.id,
              login: session.login, principal: session.principal,
              on_behalf_of: session.on_behalf_of,
              roles: session.roles, request_id: session.request_id,
              fingerprint: redactLiterals(parsed),
              literals_hash: keyedHash(auditKey, literalsOf(parsed)),
              tables: analysis.tables,
              columns: classifyColumns(analysis.columns),
              started_at: now() }
    // a statement that cannot be parsed is recorded as a refusal with
    no
    // text, never stored verbatim as a fallback
    if parsed is null:
        record.fingerprint = "unparsed"
        record.literals_hash = null
    return record
```

Las últimas líneas son las que más importan. Un fallback que guarde la sentencia en bruto cuando el parser falla conservaría justo el caso en que es más probable que aparezca un literal, porque las sentencias inusuales son las que un parser no logra procesar. No guardaría nada y dejaría que hable el rechazo. El propio almacén se clasifica al menos como Confidencial, porque describe quién mira qué, y tiene su propia lista de lectores y su propia retención. Cuánto puede registrar una empresa sobre su propio personal es una pregunta para tus responsables legales y de cumplimiento.

¿Quién audita a `ledgerly_access_admin`?

`ledgerly_access_admin` es el rol que puede crear logins y conceder cualquier cosa dentro de los roles que administra, y se usa mediante tres logins: el paso del pipeline que aplica la definición de acceso (Capítulo 3), el credential broker (Capítulo 4) y el gateway (Capítulo 7). El Capítulo 3 dijo que toda acción va al registro de auditoría; el mecanismo es que cada uno

de los tres es un productor con su propio stream, de modo que el registro nombra qué login actuó. Cada `CREATE LOGIN`, `GRANT` y `REVOKE` que emite es una entrada `event`, escrita antes del cambio, con la solicitud o la ejecución que lo causó.

Eso da una comprobación que cierra el círculo. El catálogo de la base de datos es la verdad sobre qué grants existen, y el almacén de auditoría es la verdad sobre qué se concedió. Un grant que aparece en el catálogo sin una entrada correspondiente se hizo al margen de los tres productores, y es el grant de las 2 a. m. para cuya detección se construyó la comprobación de drift del Capítulo 3. Las dos comprobaciones son una misma cosa vista desde dos extremos: la comprobación de drift compara el catálogo con el código, y esta compara el catálogo con el registro. Ninguna repara nada; ambas informan.

Migraciones

El pipeline de migraciones escribe en el almacén de auditoría antes de ejecutar una migración, como exige el Capítulo 11, y de nuevo cuando termina o falla. La razón de la entrada externa es que la herramienta de migraciones también registra lo que aplicó en una tabla dentro de `ledgerly_primary`, y `ledgerly_migrator` asume `ledgerly_owner`, que puede alterar esa tabla. Un registro que puede reescribir quien actúa es una comodidad y no evidencia. La regla es que una migración cuenta como aplicada cuando la entrada externa de inicio y el hash del commit existen en el almacén de auditoría, y una fila de la tabla de migraciones sin entrada externa correspondiente se informa mediante la misma comparación entre catálogo y registro, como un cambio que eludió el pipeline.

Alertas que alguien responderá

Un registro responde preguntas a posteriori. Merece la pena hacer algunas de esas preguntas de forma continua, y una alerta es la forma de hacerlo. La prueba que aplicaría a cada regla es si hay alguien designado para responderla y si sabe qué hacer.

Ledgerly empieza con tres reglas, cada una de las cuales aprovecha algo que el registro sabe y que las demás fuentes no.

```

function evaluateAlertRules(window):
    for each r in outcomeRecords(window):
        // 1. large reads outside the person's working hours
        if isReadClass(r.statement_class) and isHuman(r.principal)
            and (r.rows_returned > nightRowThreshold or r.outcome ==
"row-cap-closed")
            and isOutsideHours(r.started_at, r.principal):
                notify("security", "large read outside hours: " +
r.statement_id)
        // 2. break-glass in the role set without a declared request
        if "human_breakglass" in r.roles and not
hasDeclaredRequest(r):
            notify("security", "break-glass role without a
declaration: " + r.session_id)
        // 3. Restricted columns returned to a human principal
        if touchesRestricted(r.columns) and r.rows_returned > 0
            and isHuman(r.principal) and not isDeclaredBreakGlass(r):
            notify("security", "Restricted data read by a human: " +
r.statement_id)
        for each s in sessionsOpened(window):
            if "human_breakglass" in s.roles:
                notify("security", "break-glass in use: " + s.id)

```

Los umbrales son ilustrativos; el `nightRowThreshold` correcto es el que ponga ante una persona un puñado de eventos por semana. La segunda regla es la comprobación independiente de la declaración del Capítulo 8: ese capítulo alerta cuando una persona declara la emergencia, y esta regla encuentra una sesión que tiene el rol sin declaración, es decir, un grant que la eludió. La tercera regla casi nunca debería saltar. `human_readonly` no puede leer columnas Restricted y una solicitud JIT no puede nombrar ninguna, de modo que una lectura humana de datos Restricted es o bien una sesión break-glass declarada o bien un grant que no coincide con la matriz, y lo segundo es un hallazgo de diseño.

Cada regla necesita un responsable designado y una primera acción: el contacto de seguridad pregunta cuál era la tarea, quien esté on-call en plataforma confirma la declaración y abre el elemento de revisión, el equipo de plataforma revoca un grant eludido. Una regla que se da por reconocida sin mirarla es teatro de seguridad.

La revisión de accesos

Las alertas vigilan eventos. La **revisión de accesos** es la comprobación programada de que el acceso de cada principal sigue siendo necesario, y es el control que detecta el fallo lento que ningún evento muestra: un acceso que era correcto cuando se concedió y que ha dejado de serlo sin que nadie lo notara. Las personas cambian de equipo, un servicio deja de leer una tabla, un rol conserva un privilegio que su workload no ha usado en meses. El Capítulo 3 dijo que un grant incorrecto, bien declarado, pasa la comprobación de drift, y que la revisión de accesos es el lugar donde corresponde ese juicio.

La revisión compara lo que está concedido con lo que se ha usado. El registro aporta la segunda lista, y eso es lo que hace de ella una decisión y no un formulario que firmar.

```
function runAccessReview(period):
  proposals = []
  for each grant in declaredGrants():
    // role to table privileges, identity provider group to role,
    // and the logins of ledgerly_access_admin
    use = lastUse(grant, inRecords: true)
    if use is null or use.at < now() - period:
      proposals.append({ grant: grant, finding: "unused",
                        proposal: "revoke" })
    if grant.owner is null or not isActivePrincipal(grant.owner):
      proposals.append({ grant: grant, finding: "no owner",
                        proposal: "assign or revoke" })
  for each r in breakGlassUses(period):
    if reviewFor(r.id) is null:
      proposals.append({ grant: r, finding: "no review",
                        proposal: "review now" })
  // the review proposes; a named reviewer decides, and is never
  // the person whose access is under review
  return assignReviewers(proposals)
```

Tres reglas gobiernan cómo usaría yo su salida. Primera: la función propone y nunca revoca, por la razón que dio el Capítulo 3: una revocación automática de un grant del que alguien depende es una nueva forma de romper producción. Segunda: el `period` tiene que ser más largo que el ciclo más largo del workload. Un grant usado por un informe trimestral queda sin usar la mayoría de los meses, así que una revisión mensual propondría revocarlo en dos meses de cada tres. Tercera: el uso se contrasta por privilegio y no solo por tabla. Un rol que lee de una tabla cada noche y

nunca la ha actualizado es candidato a reducirse de S, U a S, y el registro tiene lo necesario porque cada registro lleva la clase de sentencia y las tablas que tocó.

La revisión tiene una segunda entrada que el registro no puede aportar: las cosas que no son grants. El registro de copias del Capítulo 13 debe estar en la agenda, y también la lista de quién puede aprobar solicitudes JIT y la lista de quién puede leer el almacén de auditoría. La pregunta para cada una es la misma: ¿la persona o el sistema sigue necesitando esto, y sigue habiendo alguien que sea responsable? El Apéndice incluye una plantilla.

Evidencia, no cumplimiento

Los regímenes de protección de datos y de tarjetas de pago suelen plantear una familia de preguntas sobre el acceso: quién puede llegar a los datos, quién hizo qué, cuánto dura el acceso, dónde están las copias y qué ocurre en una emergencia. Una auditoría bajo uno de estos regímenes normalmente querrá ver que existe un control, que se ejecutó y que alguien miró sus resultados. Tus responsables legales y de cumplimiento son quienes interpretan cualquier régimen. Esta sección no afirma más de lo que puede entregar la ingeniería descrita arriba, que es evidencia: registros que responden preguntas y una manera de demostrar que no se alteraron.

Nada de este capítulo hace que una empresa cumpla con nada. La tabla enumera las preguntas que suele hacer una auditoría, el control que responde a cada una, el artefacto que deja y el capítulo que lo construye; el Apéndice la repite como correspondencia.

Pregunta	Control	Artefacto	Capítulo
¿Quién puede llegar a los datos de producción y por qué ruta?	El gateway como única puerta, esquema de roles como código, comprobación de drift	Lista de reglas de red, definiciones de roles, informe de drift	3, 7
¿Quién ejecutó una sentencia concreta, con qué grants y qué aprobación?	<code>access_audit_record</code> con request id y aprobador	El registro y el informe de <code>verifyChain</code>	14

Pregunta	Control	Artefacto	Capítulo
¿Cuánto dura el acceso elevado y quién lo aprobó?	Solicitud JIT con aprobador y <code>expires_at</code> , job de revocación	Registros de solicitudes, log de revocaciones, comprobación de grants vencidos	8
¿Cómo se identifican y se limitan los datos sensibles?	Clasificación en el esquema, enmascaramiento, columnas retenidas	El catálogo, la verificación de la carga enmascarada	12
¿Dónde hay copias de los datos de producción y quién es responsable de cada una?	Registro con responsable y fecha de fin, sweeper	Registro, log del sweeper	6, 13
¿Quién cambia el esquema y quién lo revisó?	Pipeline de migraciones, branch protegido, entrada externa de inicio	Registros de migraciones cotejados con commits y entradas de auditoría	11
¿Sigue siendo necesario el acceso?	Revisión de accesos con revisores designados	Revisión completada con decisiones	14
¿Qué ocurre en una emergencia?	Break-glass con alerta y revisión obligatoria	Registros de revisión, historial de alertas	8, 15
¿Quién administra el acceso en sí?	Los logins de <code>ledgerly_access_admin</code> como productores separados	Sus streams y la comparación entre catálogo y registro	3, 14

Trade-offs y costos

El registro tiene una factura, y la mayor parte se paga en dos lugares.

Almacenamiento y atención. Un registro por sentencia, dos veces, crece con el tráfico, y el almacén necesita su propia retención, sus propios backups y su propia lista de lectores. El costo mayor es la atención. Un almacén que nadie consulta es teatro de seguridad caro, así que la muestra semanal de sesiones elevadas del Capítulo 7, las alertas y la revisión forman parte del precio y no son extras.

Una nueva dependencia en la ruta crítica. Como la intención se escribe antes de ejecutar la sentencia, una caída de la auditoría bloquea el trabajo privilegiado. La alternativa, ejecutar las sentencias y registrar después, hace que el registro sea opcional justo cuando más importa. Mantendría el

rechazo para los grants, el break-glass y las escrituras, y consideraría una escritura en cola para las lecturas rutinarias de `human_readonly`, con un tope a lo que puede crecer la cola, si el equipo decide que una caída de lecturas cuesta más que un hueco. Esa es una decisión que el equipo de plataforma debe tomar una vez y dejar por escrito.

Cuándo no construirlo todo: una empresa con un puñado de ingenieros, una base de datos y todavía ningún gateway no debería empezar con un servicio de append, un ancla y una cuenta separada. Empieza por tres cosas: registrar la sesión de cada persona bajo su propio login, enviar el log de la base de datos fuera del host a un lugar del que los administradores no puedan borrar, y poner a una persona designada en una revisión mensual. Añade la cadena y el ancla cuando el número de personas que podrían alterar el log haga de “confiar” una respuesta inadecuada.

Errores comunes

- **Tratar el log de la base de datos como el registro de auditoría.** Es la fuente más completa y está bajo el control de los administradores. Envíalo a otro sitio y úsalo como una entrada de la unión, no como el registro.
- **Una cadena de hashes sin un almacén separado.** La cadena parece una prueba y la puede recalcular quien pueda escribir en el almacén. Pon el almacén en otra cuenta, mantén la cadena en el servicio de append y ancla la cabeza.
- **Guardar la sentencia en bruto.** Es tentador porque facilita la investigación, y copia valores Restricted en un almacén sin protección de columnas. Guarda el fingerprint y un hash con clave, y rechaza el fallback al texto en bruto.
- **Alertas sin responsable.** Un canal que nadie tiene designado para responder se convierte en ruido. Da a cada regla un responsable y una primera acción.
- **Auditarlo todo excepto a los auditores.** El rol que concede el acceso es aquel cuyas acciones más importan. Dale sus propios productores y compara su trabajo con el catálogo.

Ideas clave

- El registro de auditoría es el registro de quién hizo qué sobre qué datos, guardado donde quien actúa no puede modificarlo, y el registro del gateway, el log de la base de datos y `audit_events` son tres entradas para él, no el registro en sí.
- Un registro de sentencia se construye a partir de una lista de preguntas escrita primero, y sus campos de identidad terminan en una persona incluso cuando actuó una máquina, mediante `principal` y `on_behalf_of`.
- La evidencia de manipulación viene de una cuenta separada, productores de solo escritura, un servicio de append que mantiene la cadena y un ancla que pertenece a otra persona, y da evidencia, nunca prevención.
- Una cadena demuestra que las entradas presentes no se alteraron, así que comprobaciones aparte buscan huecos de secuencia, intenciones sin resultado, logins de la base de datos sin origen conocido y productores que se quedaron en silencio; los tres registros se unen por el login y el request id, y la etiqueta de conexión es solo una pista.
- El registro nunca debe convertirse en una copia de los datos que describe, así que las sentencias se guardan como fingerprints con un hash con clave de los literales, y el almacén se clasifica y se restringe como los datos.
- Las alertas necesitan un responsable y una primera acción, y la revisión de accesos compara los grants con el uso durante un período más largo que el ciclo del workload, de modo que las revocaciones se pongan a partir de evidencia.
- Evidencia no es cumplimiento: la correspondencia entre pregunta, control, artefacto y capítulo muestra lo que el sistema puede demostrar, y tus responsables legales y de cumplimiento son quienes determinan qué significa esa prueba bajo cualquier régimen.

El registro ya existe, está unido y vigilado. Lo que queda es el día en que hace falta bajo presión, cuando quien pregunta ya no es un auditor sino una persona on-call con una credencial filtrada. El capítulo siguiente trata del acceso durante los incidentes y usa el registro construido aquí para averiguar qué tocó realmente un principal comprometido.

Capítulo 15 — El acceso durante los incidentes

Imagina un sábado por la tarde en Ledgerly. Sam, de soporte, escribe en el canal de on-call: le robaron un laptop de trabajo de un vehículo estacionado, y puede que hubiera una sesión del gateway abierta en ella. Robin está on-call y lee el mensaje dos veces. No se sabe que algo vaya mal, pero “puede que hubiera una sesión abierta” es el tipo de frase que hay que responder, no sopesar.

Robin necesita tres cosas, y el orden importa. La primera es detener todo lo que esa sesión, y cualquier login detrás de ella, aún puede hacer. La segunda es averiguar qué hizo mientras pudo. La tercera es mantener el negocio en marcha mientras ocurren las dos primeras, lo que significa que otras personas siguen necesitando la base de datos, algunas con urgencia, y los aprobadores habituales están en casa.

Este capítulo trata el acceso en incidentes como algo que se diseña y se ensaya. Pone a prueba los tres pilares a la vez: la autorización cuando los grants se quitan o se añaden con prisa, la ejecución cuando hay que correr reparaciones bajo presión, y la rendición de cuentas cuando el registro debe decir qué pasó. En el modelo de cinco partes, la Identidad encuentra los logins, el Grant y la Ruta los cortan, la Ejecución lleva las escrituras de emergencia y el Registro mide el daño. Este capítulo añade el último paso del estado objetivo de Ledgerly: el acceso en incidentes tiene una ruta ensayada, y la revocación es una rutina. La pregunta es cómo detiene Ledgerly el acceso, mide lo que se alcanzó y sigue funcionando, sin reabrir las puertas que cerraron los capítulos anteriores.

El problema y por qué es difícil

Un incidente es el momento en que todo control anterior parece un obstáculo. La aprobación que era razonable un martes tranquilo es una espera un sábado, el gateway que lo registra todo es una cosa más que puede es-

tar caída, y el atajo más limpio a la vista es un login que se salta todo eso. La respuesta habitual es construir de antemano ese login “para emergencias”. Es la contraseña de administrador compartida otra vez, con mejor nombre.

Tres cosas cambian bajo presión. El tiempo es corto, así que los pasos que exigen que alguien los recuerde se omiten. El conocimiento es parcial, así que la primera estimación del alcance se equivoca en un sentido o en otro. Y los actores son inusuales: a personas que rara vez tocan la base de datos se les pide de pronto que lo hagan, y las que la tocan a diario están cansadas.

La pregunta de diseño es, por tanto, qué controles se relajan y cuáles no. Mantendría una regla: la urgencia cambia la velocidad, nunca la identidad, nunca la sentencia que se ejecuta y nunca el registro. Los períodos de espera y el orden de los aprobadores pueden acortarse. Un principal que actúa con su propio login, una aprobación atada al hash de una sentencia y una entrada en el registro de auditoría, no. Si un control no puede acortarse sin eliminar una de esas tres cosas, el control está mal, no la regla.

La velocidad hay que construirla. Una revocación que requiere una docena de pasos manuales se hará en el orden equivocado o a medias. El orden es código, la consulta de alcance es código, y la conciliación que dice si el incidente dejó acceso abierto es código. Un runbook que apunta a esas piezas es un procedimiento; uno que las sustituye por criterio es una esperanza.

Contener un compromiso

Contención significa aquí detener el acceso de un principal, sea la causa un laptop perdido, una credencial filtrada o un workload que se comporta de una manera que nadie esperaba. No decide la causa. El Capítulo 4 separó dos preguntas que el pánico tiende a mezclar: cómo hacer que una credencial deje de funcionar y qué hizo mientras funcionaba. La contención responde a la primera, y el orden que sigue mantiene responsable la segunda.

El orden es: revocar los grants, cerrar las sesiones, rotar las credenciales, delimitar el daño a partir del registro de auditoría, y después registrar y notificar. Cada paso está donde está por una razón.

Revocar los grants va primero porque una sesión sin privilegios no puede hacer nada, y una reconexión no consigue nada. Cerrar las sesiones va en segundo lugar, por el trabajo en curso y las conexiones abiertas que sobreviven a un grant revocado (el Capítulo 4 hizo la misma observación sobre las credenciales vencidas). La rotación va tercera porque detiene los nuevos logins y no hace nada por los que ya están abiertos. La delimitación es de solo lectura y necesita el registro, no el acceso, así que espera hasta que el acceso haya desaparecido. La notificación va al final, pero el registro de cada paso no: cada paso se añade a medida que termina.

```
function containCompromise(incident, principal, since):
    unfinished = []
    // every step runs even if the one before failed: a stuck revoke
    must
    // not leave sessions open and the credential valid
    for each step in ["revoke-grants", "close-sessions", "rotate-
    credentials"]:
        outcome = "done"
        try:
            runContainmentStep(step, principal)
        on StepFailed:
            outcome = "failed"
            unfinished.append(step)
            notify("platform", "containment step failed: " + step + "
            " + incident.id)
        // the log never delays containment; an unreachable log queues
        the entry
        audit0rQueue({ action: "contain", actor: incident.commander,
            subject: principal, request: incident.id,
            step: step, outcome: outcome })
        // read-only and last: it needs the record, not the access
        scope = blastRadius(principal, since)
        saveIncidentRecord(incident.id, { principal: principal, since:
        since,
            scope: scope, unfinished:
        unfinished })
        notify("security", "contained, scope attached: " + incident.id)
        return unfinished
```

Dos detalles merecen una nota. La función devuelve los pasos sin terminar y no lanza una excepción, porque un principal contenido a medias es un estado que hay que seguir, y una excepción terminaría el único proceso que lo conoce. Y audit0rQueue es el mismo helper que en el Capítulo 7: una caída del log retrasa la entrada y nunca la retirada del acceso. La asi-

metría es la que eligió el Capítulo 8 para la revocación: un grant sin entrada en el log es una exposición, y una revocación sin ella es un hueco que se puede rellenar más tarde.

Lo que significa cada paso depende de qué sea el principal. La tabla recoge los cuatro casos que tiene Ledgerly.

Principal	Revocar grants	Cerrar sesiones	Rotar credenciales
Persona, logins de sesión	Quitar las tripletas JIT, <code>human_breakglass</code> y <code>human_readonly</code> de cada login de sesión que tenga la persona	<code>endSession</code> para cada login; deshabilitar a la persona en el proveedor de identidad para que no se abra ninguna sesión nueva (Capítulo 7)	Restablecer las credenciales y los factores de la persona en el proveedor de identidad; los logins de sesión se eliminan, así que no hay ninguna contraseña de base de datos que rotar
Servicio, credencial dinámica	Terminar el lease; el login se va con él (Capítulo 4)	El broker cierra las sesiones como parte de terminar el lease	Nada a mano; la siguiente solicitud recibe un valor nuevo
Servicio, credencial estática en un par A/B	Quitar de inmediato el rol <code>app_*</code> del login filtrado, sin esperar a que el login anterior se drene	Cerrar sus sesiones restantes	Publicar un valor nuevo en el login inactivo y cambiar el puntero; reiniciar los servicios que no pueden volver a leer su secret

Principal	Revocar grants	Cerrar sesiones	Rotar credenciales
Login de <code>ledgerly_access_admin</code>	Quitarle sus privilegios, lo que detiene el aprovisionamiento de cualquier principal que lo usara	Cerrar sus sesiones	Emitir una credencial nueva a ese principal; la pregunta de alcance pasa a ser qué grants y logins no explica ninguna solicitud (la comprobación de drift del Capítulo 3)

La fila estática es la que coincide con el Capítulo 4. Una credencial estática filtrada no es una rotación planificada, y el login filtrado se corta antes de que su reemplazo esté listo, lo que cuesta una caída en los servicios que lo usan. El Capítulo 4 ya aceptó ese costo, y yo mantendría la aceptación: una credencial filtrada que sigue funcionando es el problema mayor. Los servicios que necesitan un reinicio deberían estar en una lista antes del día en que importe. En el estado inicial, donde todos los servicios comparten un único login de administrador, el primer paso es la caída de toda la empresa, que es el argumento más fuerte para construir primero la revocación rutinaria.

La última fila es la pesada. El login que administra el acceso puede conceder cualquier cosa dentro de los roles que administra, de modo que contenerlo impide que el gateway, el pipeline o el broker aprovisionen nada hasta que exista una credencial nueva. Ese es el precio de cortar el login de mayor alcance, y un equipo debería decidir de antemano quién puede pagarlo.

Delimitar el radio de impacto a partir del registro

Una vez desaparecido el acceso, la pregunta pasa a ser qué hizo. La respuesta es una consulta sobre `access_audit_record`, cuyo diseño describe el Capítulo 14, ejecutada con los campos que ese capítulo define. La consulta es el radio de impacto de un principal desde el momento T en que su

acceso pudo usarse indebidamente por primera vez: cada tabla que tocó, cuántas filas devolvió, cuántas cambió y cuáles de esas tablas contienen datos Restricted.

```
// illustrative notation; field names as in Chapter 14
SELECT t.table_name,
       classOf(t.table_name)           AS data_class,
       COUNT(*)                       AS statements,
       SUM(r.rows_returned)           AS rows_returned,
       SUM(r.rows_affected)           AS rows_affected,
       SUM(CASE WHEN r.outcome = "refused" THEN 1 ELSE 0 END) AS
refused,
       MIN(r.started_at)               AS first_seen,
       MAX(r.ended_at)                AS last_seen
FROM access_audit_record r
     EXPAND r.tables AS t              // one row per table
touched
WHERE r.kind = "outcome"              // one row per statement;
intent rows have no counts
  AND r.login IN :logins
  AND r.started_at >= :since
GROUP BY t.table_name
ORDER BY (data_class = "Restricted") DESC, rows_returned DESC;
```

Elegir `:logins` y `:since` es donde reside la mayor parte del criterio. Para una persona, el gateway crea un login nuevo por cada sesión, así que el conjunto es cada login de sesión que el principal haya tenido desde T, no uno solo. Para T, usa el primer momento en que el acceso pudo usarse indebidamente, como la hora en que se sabe por última vez que el laptop estaba en manos de su propietario, y no la hora en que alguien lo notó. Una ventana demasiado estrecha produce una respuesta limpia a la pregunta equivocada.

El resultado vale lo que vale el registro, así que tres comprobaciones van antes de que nadie lo lea.

1. `verifyChain` pasa sobre la ventana, tal como lo define el Capítulo 14. Si falla, el resultado es evidencia de algo distinto del incidente, y el incidente recibe un segundo registro.
2. La ventana no tiene huecos de secuencia ni sesiones del gateway sin sesión en la base de datos, ni lo contrario, que es la detección de huecos del Capítulo 14. Un hueco dentro de la ventana es un tramo en el que la respuesta se desconoce.

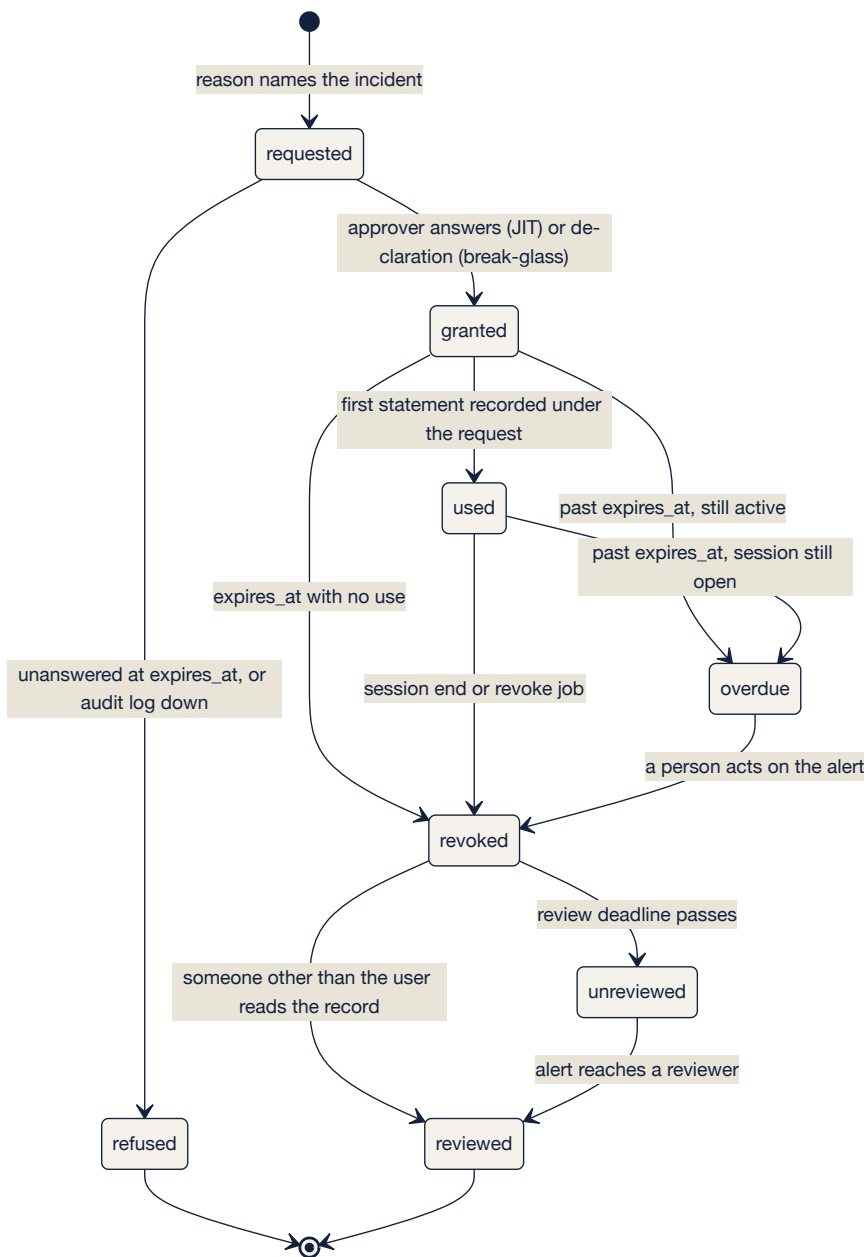
3. Las rutas laterales del principal se comprueban aparte: las restauraciones, los exports y las copias que solicitó (Capítulo 13), que se encuentran mediante el registro de copias y no mediante esta consulta.

Si falla alguna comprobación, el resultado es una cota inferior, y yo lo diría en el registro del incidente con esas mismas palabras. La consulta tiene además sus propios límites. `rows_returned` es un conteo, no una lista: te dice que una tabla Restricted devolvió cierto número de filas en la ventana, y no puede decirte de qué clientes eran. Los campos `fingerprint` y `literals_hash` permiten preguntar si se repitió una sentencia conocida, pero los literales están redactados por diseño. Así que el registro acota la exposición y no la enumera. Un `rows_returned` distinto de cero en una tabla Restricted es la línea que necesitan los contactos de seguridad y tus responsables legales y de cumplimiento, y la interpretación de lo que significa, incluido cualquier deber de notificación, les corresponde a ellos.

La vida del acceso durante un incidente

La contención quita acceso. Un incidente también necesita que se añada acceso: alguien tiene que mirar, y alguien puede tener que reparar. El riesgo en esa dirección es el grant creado con prisa que sigue ahí el lunes. La cura es un ciclo de vida con un final visible, en el que todo acceso creado para un incidente pasa por los mismos cinco estados.

El diagrama es una vista sobre los registros que el Capítulo 8 ya lleva, no un campo de estado nuevo. `granted` es `approved` en el registro de la solicitud, `used` significa que se registró una sentencia bajo la solicitud, y `reviewed` significa que alguien distinto de quien lo usó leyó el registro. Fíjate en las dos salidas que no llegan a `reviewed`: un grant que permanece más allá de su fin y una revisión que nunca ocurre.



Para que esto funcione, toda solicitud hecha durante un incidente lleva el identificador del incidente en su motivo. El formulario del gateway puede exigirlo, y una solicitud sin él durante un incidente abierto se rechaza. Esa convención es la que permite que un job pregunte “todo lo que concedió este incidente”.

```

function reconcileIncidentAccess(incident):
    // watches the incident's own requests; never changes access
    (Chapter 8:
    // an automatic revoke of unknown grants is a way to break
    production)
    for each r in requestsWithReason(incident.id):
        if r.status == "approved" and r.expires_at < now() -
        graceMinutes:
            notify("platform", "incident grant past expiry: " + r.id)
            if r.status == "revoked" and reviewFor(r.id) is null
                and now() - r.expires_at > reviewDeadline:
                    notify("security", "incident access not reviewed: " +
                    r.id)
            // the database side: grants that no request explains
            for each g in unexplainedGrants(since: incident.opened_at):
                notify("platform", "grant with no request: " + g.login)

function closeIncident(incident):
    reconcileIncidentAccess(incident)
    // an incident is not over while any access it created is still
    open
    if openFindings(incident.id) > 0:
        raise IncidentHasOpenAccess
    markClosed(incident.id)

```

Este job no sustituye la comprobación de vencidos ni la de drift de los capítulos 8 y 3. Mira los mismos hechos desde el lado del incidente, y le da al incidente una compuerta: nadie puede marcarlo como cerrado mientras un grant siga vivo o un uso de break-glass esté sin revisar. Un equipo que use un sistema de tickets puede conectar `closeIncident` al ticket, lo que convierte “se nos olvidó revocar” en un bloqueo visible.

Revisado

El último estado es el que más a menudo se omite, porque el incidente parece terminado. La revisión es la que describe el Capítulo 8 para el break-glass, extendida a todos los grants que hizo el incidente. Es sin culpas y pregunta por el sistema: ¿fue el acceso el más pequeño que podía hacer el trabajo, y existía una ruta más rápida que debería haber existido? El Apéndice incluye una plantilla posincidente. El hallazgo que rinde suele ser una ruta que faltaba.

Escrituras durante un incidente

La reparación es el lugar más tentador para relajar un control, porque el daño se ve y los datos están mal ahora mismo. El Capítulo 10 dejó establecido que una reparación es una corrección de datos, y el Capítulo 8 dejó establecido que el break-glass no es una vía para reparaciones. Bajo presión mantendría ambas decisiones y haría rápida la ruta conforme, en lugar de hacer disponible la ruta equivocada.

La primera elección es a qué ruta pertenece la necesidad. El orden de las comprobaciones va de la puerta más estrecha a la más ancha.

```
function chooseIncidentWritePath(need):
    // first match wins; Restricted values are never written by a
    person
    if need.touches_restricted_column or need.is_owner_link_change:
        return "product path: admin-console asks payments-service
(Chapters 9, 10)"
    if need.is_schema_change:
        return "pipeline, expedited lane (Chapter 11)"
    if need.stops_ongoing_harm and need.is_small_limiting_action:
        // pausing, disabling or flagging: never the repair itself
        return "break-glass session, reviewed afterward (Chapter 8)"
    return "data fix: JIT request, runner, expedited review (Chapters
8, 10)"
```

El break-glass solo es accesible para la pequeña acción limitante: un flag que impide que un job escriba filas incorrectas, un estado que retiene un lote. La reparación duradera de los datos es siempre una corrección de datos, con el mismo razonamiento de antes: una reparación ejecutada en una sesión break-glass en bruto se salta el único control que lee la sentencia antes de ejecutarla, y se ejecuta donde todavía es posible un `UPDATE` sin límites (el Capítulo 8 lo nombra como un riesgo aceptado). Durante un incidente la tentación es mayor, y también lo es el costo de una escritura equivocada.

La corrección de datos conserva su forma. La solicitud nombra una corrección revisada y lleva un `approved_hash` tomado del registro del merge, el runner rechaza un script cuyo hash difiera, el grant se deriva de la corrección, y la protección de conteo y el snapshot siguen aplicándose (capítulos 9 y 10). Lo que el incidente cambia es la revisión.

```

function expediteFixReview(fix, incident, reviewer):
    // relaxes: the waiting period, and who may be asked first
    // never relaxes: a reviewer who is not the author, the hash, the
runner
    if incident.status != "open":
        raise NoOpenIncident
    if reviewer == fix.author:
        raise SelfApprovalNotAllowed
    mergeRecord = mergeFix(fix.id, reviewedBy: reviewer, expedited:
true,
                        incident: incident.id)
    // the hash is read from this record by the gateway, not from the
runner
    queueFollowUpReview(fix.id, reviewer: "platform", due:
nextWorkingDay)
    notify("platform", "expedited fix merged: " + fix.id)
    return mergeRecord

```

Esto refleja el lane expedited que el Capítulo 11 da a las migraciones: el mismo job, la misma credencial, el mismo registro, distinto momento. La revisión de seguimiento es donde se paga el atajo, y es también lo que convierte una ruta acelerada en un evento contado. Si un equipo descubre que la mayoría de las correcciones de un trimestre la usan, el lane normal es demasiado lento para el trabajo, y la respuesta correcta es acelerar el lane normal, no dejar de contar.

Advertencia: Una corrección que “necesita” break-glass es una señal sobre otra cosa, como dice el Capítulo 10: un techo al que le falta una columna, un grupo de aprobadores demasiado pequeño para esa hora o una columna Restricted que pertenece a la ruta de producto. En un incidente esa señal es fácil de ignorar y cara de ignorar dos veces.

Los casos que los capítulos anteriores dejaron pendientes

Tres situaciones fueron aplazadas por capítulos anteriores. La tabla es la decisión por adelantado; las subsecciones dan el razonamiento.

Situación	Qué se detiene	Qué no cambia	Decidido de antemano
El gateway está caído	El acceso humano a los datos (no hay segunda puerta)	Los servicios siguen funcionando con sus propias credenciales	Quién restaura el gateway, y cuándo puede usarse el instrumento sellado
Hace falta una restauración durante el incidente	Nada, si la ruta se ensayó	Aprobador, destino aislado, entrada en el registro, borrado programado	Un aprobador designado, un destino preparado, un runbook de restauración
Las aprobaciones son lentas por la noche	Una reparación que no es una emergencia	La autoaprobación sigue rechazada	Un orden de escalada y quién puede ampliar el grupo

El gateway está caído

Los capítulos 7 y 8 decidieron que no hay una segunda puerta: las personas llegan a la base de datos solo a través del gateway, de modo que una caída del gateway es también una caída del break-glass. El razonamiento allí fue que una credencial que funciona sin el gateway funciona también sin el logging, sin la identidad y sin el vencimiento. Me mantendría en eso en la peor noche, y el primer movimiento en una caída es fijarse en lo que no se ha detenido. Los servicios se conectan con sus propias credenciales y siguen funcionando, así que muchos incidentes en los que el gateway resulta estar caído no necesitan que una persona esté en la base de datos. Revertir el deploy, pausar el job, cambiar el feature flag, escalar el servicio: nada de eso pasa por el gateway.

Si el incidente sí necesita acceso a datos, el incidente son ahora dos incidentes, y el primero es el gateway. Restaurarlo va antes que la cuestión de los datos, y quienes lo restauran son el equipo de plataforma, usando la ruta de deploy y la redundancia que pidió el Capítulo 7.

Si todas las instancias están caídas, no existe ninguna ruta humana a los datos, y el único instrumento que queda es el superusuario que el Capítulo 3 trató como sellado. Es una herramienta a nivel de motor, no una ruta de datos: elude las protecciones de solo-append y de Restricted, y su ruta de red está cerrada a las personas. Limitaría su uso a restaurar el gateway o aquello de lo que depende el gateway, nunca a leer ni escribir datos de negocio, y dejaría la regla por escrito.

Advertencia: No uses el instrumento sellado para mirar datos porque el gateway esté caído. Elude las protecciones de solo-append y de Restricted, y el acceso que da es el acceso que todos los capítulos anteriores se construyeron para impedir. Si hay que leer datos, restaura primero el gateway.

Su uso bajo la regla tiene un mecanismo. La obtención de la credencial deja un registro, hay una segunda persona presente y nombrada, abrir la ruta de red es un cambio con responsable y hora de fin, y la credencial se rota después porque una persona la ha visto. El gateway no está registrando durante esta ventana, así que el log del motor es el único rastro. Cuando el gateway vuelve, el equipo de plataforma añade una entrada a mano, como la entrada de auditoría externa que describe el Capítulo 14 para las migraciones, y el uso recibe la revisión que recibe todo uso de break-glass.

Una restauración durante un incidente

El Capítulo 13 hizo de la restauración una solicitud: un aprobador que no es quien la solicita, un destino aislado, una entrada en el registro, un borrado programado y un inicio cerrado. El costo que se nombró allí fue la velocidad durante un incidente, y su respuesta fue ensayar y arreglar la aprobación de antemano. Ese es todo el mecanismo, así que conviene precisar qué significa “de antemano”.

Antes del incidente, el runbook nombra quién puede aprobar una restauración por la noche, el destino está preparado y aislado, y la solicitud de restauración tiene un formulario que admite un identificador de incidente. Durante el incidente, la solicitud sigue su curso sin cambios: el aprobador no es quien la solicita, la entrada en el registro va primero, el borrado programado se fija antes de que exista la copia, y los logins se deshabilitan y se aplica la definición de acceso antes de que la copia se abra. El incidente aporta el motivo, no un atajo.

Para qué sirve la copia importa más que cómo se hace. Durante un incidente una copia restaurada es una fuente de lectura, normalmente de valores antiguos: el estado de una columna antes de una escritura equivocada, o las filas eliminadas por error. Sus lectores son las personas de la solicitud. La reparación que sigue se ejecuta contra el primary como una corrección de datos, con los valores leídos de la copia como entrada de un

script revisado, y el `reapplyDeletions` del Capítulo 13 se ejecuta antes de que la copia se abra, para que la copia no traiga de vuelta datos que se borraron a propósito. Reemplazar el primary por una copia restaurada es una decisión de recuperación, y pertenece al equipo de plataforma y al incident commander. La cuestión de acceso dentro de ella es la de arriba: quién lee la copia y cuándo se borra.

Aprobaciones lentas por la noche

El escenario nocturno del Capítulo 8 terminó en una decisión que este capítulo mantiene: si el grupo de aprobadores no responde y la necesidad puede esperar, deja una nota y detente. Ese es un buen resultado, y uno que el Ledgerly de antes no podía producir. Para una necesidad que no puede esperar, la escalada está integrada en la solicitud, de modo que un page sin respuesta baja por el grupo dentro de la misma solicitud.

```
function escalateApproval(request, pool):
  // one request, one record: the page moves, the request does not
  restart
  for each approver in pool.inOrder():
    if approver == request.requester or approver ==
request.fix_author:
      continue
    page(approver, request.id)
    if waitForAnswer(request.id, within: escalationMinutes):
      return
  // pool exhausted, window still open: the incident commander
  decides
  notify("platform", "approval unanswered, pool exhausted: " +
request.id)
  // never widens the pool or approves by default; the request stays
  pending
  // until expires_at, when it is refused
```

Dos reglas rodean esto. Primera: el grupo solo lo amplía quien es responsable de él, y la ampliación queda registrada: durante un incidente declarado, el incident commander puede añadir un aprobador designado para ese incidente, con una fecha de fin. Segunda: sigue sin construirse una aprobación automática para quien esté on-call, porque un control que concede toda solicitud en cuestión de segundos no es un control. Si las aprobaciones superan con regularidad la ventana del grant, el hallazgo tiene que ver con la rotación de on-call, el tamaño del grupo o la hora, y el remedio es añadir personas al grupo, no sacar al aprobador del circuito.

Ensayar las rutas

Una ruta que nunca se ha ejecutado no está probada. Todo el capítulo se construye con pasos que nadie quiere descubrir por primera vez durante un incidente, así que cada uno necesita un ensayo, llamado aquí `game day`: un ejercicio programado con un escenario, un resultado esperado y una comprobación sobre el registro. Cuatro escenarios cubren lo que construyó este capítulo. Son defensivos por diseño, cada uno parte de un disparador que el monitoreo ya sabe ver, y cada uno lleva una condición de éxito observable.

Esce- nario	Disparador	Comportamiento esperado	Condición de éxito	Partes ejerci- tadas
Filtra- ción de creden- cial	La credencial de un prin- cipal de prueba se declara expuesta	<code>containCompromise</code> en orden; consulta de alcan- ce adjunta al incidente	Ningún lo- gin activo para el principal, <code>unfinished</code> vacío, el al- cance nom- bra todas las tablas que tocó la prueba	Identi- dad, Grant, Regis- tro
Gate- way caído	Una y luego todas las ins- tancias del gateway dete- nidas en una copia que no es de producción	El acceso humano se de- tiene; los servicios siguen funcionando; se ejecuta la ruta de restauración	El equipo de platafor- ma restaura el gateway con el run- book; ins- trumento sellado sin usar, o usa- do bajo su regla con un registro	Ruta, Regis- tro
Export sospe- choso	Una alerta por un <code>rows_returned</code> grande, de <code>evaluateAlertRules</code> (Capítulo 14)	Contener al principal; en- contrar el destino en el registro de copias; borrarlo	La copia se encuentra, se termina y se regis- tra; la con- sulta de al- cance coin- cide con el registro	Identi- dad, Regis- tro, Ejecu- ción

Esce- nario	Disparador	Comportamiento esperado	Condición de éxito	Partes ejerci- tadas
UPDATE masivo equivocado	Una escritura que cabe en el row cap y cambia las filas equivocadas, en un entorno de ensayo	Detener nuevas escritu- ras; calcular la reversión a partir del snapshot; eje- cutarla como una correc- ción de datos acelerada	Reversión fusionada por un revi- sor que no es el autor; revisión de seguimien- to en cola	Ejecu- ción, Grant, Regis- tro

Un game day se supera solo con el resultado observado, nunca porque alguien recuerde qué hacer. Si una prueba termina porque alguien conocía un atajo, ha fallado, y el atajo pasa a ser lo que hay que construir. Anota los tiempos que tardó realmente cada paso; esos números pertenecen a tu empresa, y yo no daría una cifra que no sea la suya.

Usa una copia que no sea de producción siempre que sea posible, y principales sintéticos en los ejercicios en producción, de modo que no se lean datos reales de clientes para practicar la lectura. Las copias de ensayo se registran y se cierran como cualquier otra copia (Capítulo 13).

Trade-offs y costos

El primer costo es el propio ensayo. Los game days apartan a los ingenieros del trabajo de producto, y una prueba necesita un responsable y un calendario. Para un equipo de tres, una sola prueba al año que cubra la filtración de credencial es un comienzo razonable, y los otros tres escenarios pueden esperar a que el equipo crezca.

El segundo costo es la caída que causa la contención. Revocar de inmediato la credencial de un servicio es una caída en ese servicio, y contener `ledgerly_access_admin` detiene el aprovisionamiento. Ambas son elecciones que hay que hacer de antemano, con las personas que las soportan, porque quien decide a las 3 a. m. no debería estar además descubriendo lo que cuesta la elección.

El tercer costo es la convención del identificador de incidente. Añade un campo a cada solicitud y un motivo más para que se rechace un formulario, y solo vale lo que valga la disposición de la gente a rellenarlo. La con-

ciliación que depende de él falla en silencio cuando no se rellena, así que el gateway debería imponer el campo y no confiar en el hábito.

El cuarto costo es la honestidad sobre el alcance: algunos incidentes terminan con una cota mayor que la verdad, y yo informaría de la cota en lugar de afirmar una precisión que el registro no tiene.

No apliques toda la maquinaria a un sistema que todavía no tiene registro de auditoría. `blastRadius` necesita el registro, y `reconcileIncidentAccess` necesita los registros de solicitudes; un equipo que no los tiene debería construirlos primero (capítulos 8 y 14) y, mientras tanto, escribir un runbook corto que nombre quién revoca, quién cierra las sesiones y quién es responsable de la pregunta sobre qué se tocó.

Errores comunes

- **Rotar sin revocar.** Una rotación cambia el secret y deja en su sitio la sesión abierta y los privilegios concedidos. Es tentadora porque la rotación es una rutina que el equipo ya tiene. Revoca primero, luego cierra las sesiones, luego rota.
- **Delimitar antes de contener.** El impulso de entender llega antes que el de detener, y un equipo puede pasar una hora con la consulta mientras el acceso sigue vivo. Contén primero. El registro sobrevive a la contención porque vive en un almacén que el acceso no puede tocar.
- **Fiarse del log sin comprobarlo.** Una consulta de alcance sobre una ventana con un hueco se lee como una respuesta limpia. Comprueba primero la cadena y la secuencia, e informa de una cota inferior cuando falle alguna.
- **Break-glass como lane de reparación.** Es rápido y está disponible, y no tiene revisión de sentencias. Úsalo para la pequeña acción limitante y ejecuta la reparación como una corrección de datos acelerada.
- **Dejar abierto el acceso del incidente.** El incidente parece terminado cuando se arregla la causa, y los grants se quedan. Ata el cierre del incidente a la conciliación, de modo que el ticket no pueda cerrarse mientras haya acceso abierto.

Ideas clave

- La urgencia cambia la velocidad de un control, nunca la identidad que actúa, la sentencia que se ejecuta ni el registro que se deja.
- La contención tiene un orden: revocar los grants, cerrar las sesiones, rotar las credenciales, delimitar a partir del registro de auditoría, y luego registrar y notificar, y cada paso se ejecuta aunque el anterior haya fallado.
- Revocar no es rotar: una rotación detiene los nuevos logins y no hace nada por los que ya están abiertos, por eso una credencial estática filtrada se corta primero y se reemplaza después.
- El radio de impacto sale de una consulta sobre `access_audit_record`, y es una cota inferior siempre que la cadena o la secuencia tengan un hueco.
- El acceso de un incidente pasa por solicitado, concedido, usado, revocado y revisado, y un incidente no puede cerrarse mientras siga abierto algún grant que creó.
- Las reparaciones durante un incidente son correcciones de datos con una revisión acelerada, y el break-glass se reserva para la pequeña acción que detiene el daño.
- Una ruta que no se ha ensayado es una suposición, así que la filtración de credencial, la caída del gateway, el export sospechoso y el `UPDATE` masivo equivocado tienen cada uno un game day con una condición de éxito observable.

El Capítulo 16 reúne los controles de todo el libro en una lista de verificación y una escalera de madurez, y vuelve a la frase a la que ha servido cada capítulo.

Capítulo 16 — Una lista de verificación de acceso y reflexiones finales

Piensa en Ledgerly un año después del cuestionario de seguridad del Capítulo 1. Otro cliente potencial envía las mismas dos líneas: “Enumeren a cada persona y cada sistema que puede conectarse a su base de datos de producción” y “¿Cómo saben lo que hicieron?”. Alex abre el inventario de accesos en lugar del gestor de contraseñas, lee la lista de puertas y la devuelve en una tarde. Nada en la respuesta es impresionante. Es corta, cada fila tiene un responsable y una fecha de fin, y nada de ello exigió rastrear.

Robin, que está on-call esa semana, pregunta por el chat: “Tenemos un gateway, un runner, un pipeline, un almacén de auditoría y tres jobs de revisión. ¿Cuáles de estos necesitábamos y cuáles construimos porque lo decía el libro?”. Alex no tiene una respuesta ordenada. Algunas piezas se ganaron su lugar la primera vez que un grant venció por sí solo; otras nunca se han usado en una situación real, y Alex no sabe decir si eso significa que funcionan o que están ociosas.

Este capítulo sirve a los tres pilares a la vez: autorización, ejecución y rendición de cuentas. No añade ningún control nuevo. Añade las etapas del estado objetivo en el orden en que un equipo puede alcanzarlas, los patrones puntuados lado a lado y una lista de verificación cuyas líneas son observaciones y no opiniones. La pregunta que responde es la de Robin: ¿cómo decide un equipo qué construir a continuación y cómo sabe que un control funciona?

El problema y por qué es difícil

Quince capítulos dejan al lector con muchos controles y sin orden. Cada uno se presentó con su propio razonamiento, y cada razonamiento era sólido en su capítulo. Leídos en conjunto parecen un proyecto de un año o

más, y un equipo ante esa lista suele hacer una de dos cosas. Empieza por lo más barato y se detiene cuando el trabajo se vuelve político, o empieza por lo más aterrador y construye una pieza elaborada del diseño mientras la contraseña compartida sigue en cinco archivos de configuración.

La dificultad es que en el trabajo de acceso es difícil ver cuándo algo está “terminado”. Un esquema de roles que existe en el papel y uno que se aplica se ven igual en una reunión de revisión, así que los equipos miden por lo que se instaló, e instalado es la unidad equivocada. La unidad útil es una prueba que alguien pueda ejecutar: una sentencia que pueda rastrear, una conexión que se rechaza, un grant que ya no está.

También es una cuestión de cómo está construido el sistema y no de lo que dice la política. Una política enuncia lo que debería ocurrir; un diseño determina lo que puede ocurrir y qué deja registro cuando ocurre. Cada línea de más abajo se puede responder mirando un sistema, un log o una tabla, y una línea que solo se puede responder preguntando a alguien qué piensa hacer no está en la lista.

La tesis, aterrizada

El libro defendió una sola frase, y aquí se contrasta con los capítulos:

El acceso a bases de datos en producción no es un problema de distribución de credenciales; es un problema de autorización, ejecución y rendición de cuentas.

La frase tiene tres pilares, y cada pilar se apoya en capítulos concretos. La autorización es grant más ruta, decididos por solicitud. La ejecución es lo que le ocurre a cada sentencia. La rendición de cuentas es identidad más registro. Una contraseña toca solo el primero de ellos, y solo en la puerta.

Pilar	Partes del modelo	Capítulos que lo construyen	La pregunta que el lector ya puede responder
Autorización	Grant, Ruta	3, 5, 6, 8, 12	¿Qué puede hacer este principal, dónde, durante cuánto tiempo y con qué columnas ocultas?
Ejecución	Ejecución	7, 9, 10, 11, 13	¿Cómo llegó esta sentencia a los datos, quién o qué la inspeccionó y qué la acotó?

Pilar	Partes del modelo	Capítulos que lo construyen	La pregunta que el lector ya puede responder
Rendición de cuentas	Identidad, Registro	4, 7, 14, 15	¿Qué persona o instancia de workload actuó, a solicitud de quién y qué evidencia la une con el resultado?

Los capítulos 1 y 2 enmarcan los tres, y el Capítulo 15 los pone a prueba bajo presión. El Capítulo 7 aparece dos veces porque el access gateway es a la vez la ruta y el lugar donde una persona se convierte en un login de sesión que un log puede nombrar; el Capítulo 13 está bajo ejecución porque el paso de export de `reporting-job` ejecuta un conjunto de sentencias, y las demás puertas laterales son rutas para esquivar los mismos controles.

Por qué el trabajo sobre las credenciales era necesario y no era la respuesta

El Capítulo 4 trata de las credenciales, y es el capítulo que la mayoría de los lectores espera que sea el centro del libro. Es necesario. Un servicio que lee su contraseña de un archivo compartido no tiene identidad propia, y sin identidades distintas el registro no puede nombrar a nadie. Las credenciales dinámicas y de corta duración convierten la revocación en un paso rutinario. La workload identity elimina el archivo.

Nada de eso decide qué puede hacer un principal una vez dentro, cómo se comprueba una sentencia ni qué se deja por escrito. Un equipo podría terminar el Capítulo 4 a la perfección y aun así tener un rol que es propietario de todas las tablas, una sesión desde un laptop que hace commit de cada sentencia tal como se escribió y un log en el que cada línea tiene un nombre distinto y ninguna tiene un motivo. Un almacén de contraseñas aporta los nombres distintos que necesita la parte Identidad y, mediante la revocación, un fragmento de Grant; una contraseña toca solo el primer pilar, y solo en la puerta. Las demás partes se construyen en otro lugar, y el resto del libro dice dónde.

Cuando una propuesta diga “arreglaremos el acceso mejorando X”, ubica X en el modelo antes de aprobarla:

```
function placeControl(proposal):
    parts = partsOfModelTouched(proposal)    // Identity, Grant, Path,
    Execution, Record
    if "Execution" not in parts and "Record" not in parts:
        // a control that touches only the door leaves two of three
        pillars as they were
        note("improves how access is obtained; says nothing about what
        it does or what remains")
        for each part in ["Identity", "Grant", "Path", "Execution",
        "Record"]:
            if part not in parts:
                note("unchanged: " + part)
    return notes
```

La función no es un gate. Nombra qué partes deja intactas una propuesta, porque una parte vacía es un hallazgo y no una omisión.

Un catálogo de patrones, puntuados por pilar

El Capítulo 1 dibujó el abanico de formas de entrar, y el Capítulo 7 comparó las rutas concretas. La tabla siguiente pone lado a lado los cinco patrones de ese abanico y puntúa cada uno frente a los tres pilares. Las puntuaciones son mi criterio, no una medición. Dicen cuánto de un pilar puede evidenciar el patrón por sí solo, con los grants y el logging de los capítulos anteriores en igualdad de condiciones. Lee “fuerte” como “el patrón está construido para demostrar esto”, “parcial” como “demuestra una parte, o solo si se añade algo más”, y “ninguna” como “el patrón guarda silencio aquí”.

Patrón	Autorización	Ejecución	Rendición de cuentas
Acceso directo (del cliente a la base de datos)	Parcial: solo grants, el alcance es el que permite la red	Ninguna	Parcial en el mejor de los casos: logins por persona si alguien los provisiona, de lo contrario ninguna
Bastion	Parcial: reduce la ruta de red	Ninguna: reenvía, no distingue un <code>SELECT</code> de un <code>DROP</code>	Parcial: quién inició sesión en el host, no qué se ejecutó
Session proxy o access gateway	Fuerte: logins por sesión, vencimiento, grants por solicitud	Parcial: acotada por los grants y los guarded settings, las sentencias se escriben libremente	Fuerte en quién y cuándo, parcial en qué

Patrón	Autorización	Ejecución	Rendición de cuentas
Ejecutor que media las sentencias	Fuerte	Fuerte: se inspecciona antes de ejecutarse, vinculada a un approved hash	Fuerte: sentencia, aprobador y resultado en un solo registro
Ruta de producto (sin SQL)	Fuerte: la operación es el permiso	Fuerte: la operación es fija	Parcial: la persona queda en el rastro de auditoría de la herramienta, el log de la base de datos muestra el login del servicio

Acceso directo

- **Demuestra:** que existe un login y que sus grants eran los que eran.
- **No puede demostrar:** qué persona usó un login compartido, ni qué hizo una sentencia.
- **Costo:** ninguno para construirlo, que es su atractivo. Todos los costos llegan después, como una semana de rastreo.
- **Modo de fallo:** la contraseña de administrador compartida del Capítulo 1.
- **No lo uses cuando:** más de una persona o workload pueda alcanzar el login. Es tolerable en un sistema de un solo ingeniero cuyo único consumidor es su responsable, y yo dejaría por escrito qué haría que eso dejara de ser cierto.

Bastion

- **Demuestra:** que una conexión pasó por un único host controlado y, si las cuentas en él son por persona, quién inició sesión allí.
- **No puede demostrar:** cuál fue cualquier sentencia, ya que el host no habla el protocolo de la base de datos.
- **Costo:** un host que parchear y mantener disponible, a cambio de un solo lugar donde cortar la ruta de red.
- **Modo de fallo:** una cuenta de bastion compartida convierte el host otra vez en la contraseña compartida, un salto antes.
- **No lo uses cuando:** necesites evidencia a nivel de sentencia. Sigue siendo útil como alcance de red delante de algo que sí registra.

Session proxy o access gateway

- **Demuestra:** que una persona con nombre tuvo un login de sesión específico, entre dos marcas de tiempo, bajo grants específicos.
- **No puede demostrar:** que una sentencia escrita en esa sesión fuera buena idea. La sesión es un handout (Capítulo 9) y se puede escribir cualquier cosa que los grants permitan.
- **Costo:** un sistema nuevo que debe estar disponible cuando alguien lo necesita, y la ruta de aprobación que lo acompaña (los capítulos 7 y 8).
- **Modo de fallo:** un gateway con una segunda puerta a su lado, que es una contraseña compartida con mejor nombre.
- **No lo uses cuando:** las tareas se repiten y tienen una forma fija. Pertenecen a una ruta de producto, donde nadie necesita una sesión.

Ejecutor que media las sentencias

- **Demuestra:** que este conjunto exacto de sentencias, identificado por su hash, fue aprobado por una persona con nombre, fue inspeccionado, se ejecutó dentro de un límite y produjo un resultado contado.
- **No puede demostrar:** que el análisis entendiera la sentencia. El análisis de consultas reduce errores y no es el límite de autorización, que sigue en los grants (Capítulo 9).
- **Costo:** cada tarea espera una revisión, nada es interactivo y alguien tiene que construir y mantener el runner.
- **Modo de fallo:** una aprobación que siempre se concede en segundos, que es teatro de seguridad con pasos adicionales.
- **No lo uses cuando:** el trabajo sea exploratorio. Una persona que intenta entender un estado necesita primero una sesión de lectura, y para eso existe `human_readonly`.

Ruta de producto

- **Demuestra:** que se ejecutó una operación definida, para un rol en una herramienta, con el rastro de auditoría propio de la herramienta.
- **No puede demostrar:** nada sobre las tareas para las que nadie construyó una funcionalidad. Las preguntas de forma libre no tienen ruta.

- **Costo:** una funcionalidad por tarea, y el ciclo de cambio más lento de los cinco.
- **Modo de fallo:** la ruta de producto que acumula una salida de emergencia de “ejecutar cualquier SQL” porque el primer caso urgente no tenía funcionalidad.
- **No lo uses cuando:** la tarea todavía no se haya repetido. El Capítulo 10 describe la señal para convertir una corrección repetida en una funcionalidad, y una tarea que ha ocurrido una vez no es esa señal.

El patrón de más a la derecha no siempre es el mejor, y un ejecutor que media las sentencias es más de lo que necesita una pregunta sobre las facturas de un solo cliente. Elegiría por tarea, con el patrón más barato que aún evidencie los pilares que la tarea pone en riesgo:

```
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
    (Chapter 11)
    if task.changesSchema:
        return "migration pipeline"
    // Restricted columns and owner links are changed by the product,
    not by SQL
    if task.writesRestrictedColumn or task.changesOwnerLink:
        return "product path"
    if task.writesData:
        // reviewed statement set, approved hash, runner executes
        return "statement-mediating executor"
    if task.readsData:
        return "access gateway: human_readonly on ledgerly_replica_2"
    // anything else is an exception with a named owner and an end
    return "exception: owner, reason, ends_on"
```

La última línea es lo esencial. Una ruta que no encaja entra en el inventario con un responsable y una fecha de fin, como pide el bloque de inventario del Capítulo 1, y una excepción sin ambos es un resto.

La arquitectura de referencia

El diagrama muestra el estado objetivo de Ledgerly con todos los controles del libro en una sola imagen. Fíjate en tres cosas: que ninguna flecha humana llega a una base de datos salvo a través del gateway, que toda escri-

The diagram illustrates the Zero Trust Network Architecture (ZTNA) solution components and their interactions. The components are represented by boxes (Person, Identity provider, Access gateway, Data fix runner, Services, Credential broker, Review jobs, Pooler, Migration pipeline: ledgery_migrator) and cylinders (ledgery_replica_2, ledgery_primary, Append-only audit store, separate account). The interactions are labeled with terms like 'statements', 'revoke', 'append', 'dynamic logins', 'direct connection: refused by network reach', and 'writes forwarded from the runner only'.

```

graph LR
    Person --> Identity_provider[Identity provider]
    Identity_provider -- "direct connection: refused by network reach" --> Access_gateway[Access gateway]
    Identity_provider -- "writes forwarded from the runner only" --> ledgery_primary[(ledgery_primary)]
    Access_gateway -- "statements" --> Data_fix_runner[Data fix runner]
    Data_fix_runner -- "revoke" --> Services[Services]
    Services -- "dynamic logins" --> Credential_broker[Credential broker]
    Credential_broker -- "append" --> Review_jobs[Review jobs: revoke, alert rules, access review]
    Review_jobs -- "append" --> Pooler[Pooler]
    Pooler -- "append" --> Migration_pipeline[Migration pipeline: ledgery_migrator]
    Migration_pipeline -- "append" --> ledgery_replica_2[(ledgery_replica_2)]
    Migration_pipeline -- "append" --> ledgery_primary
    Migration_pipeline -- "append" --> Audit_store[(Append-only audit store, separate account)]
    ledgery_replica_2 -- "read session: human, read-only" --> ledgery_primary
    ledgery_replica_2 -- "append" --> Audit_store
    ledgery_replica_2 -- "append" --> separate_account[(separate account)]
    ledgery_replica_2 -- "append" --> ledgery_primary
    ledgery_replica_2 -- "append" --> Migration_pipeline
    ledgery_replica_2 -- "append" --> Pooler
    ledgery_replica_2 -- "append" --> Review_jobs
    ledgery_replica_2 -- "append" --> Services
    ledgery_replica_2 -- "append" --> Credential_broker
    ledgery_replica_2 -- "append" --> Data_fix_runner
    ledgery_replica_2 -- "append" --> Person
  
```

Cada caja se corresponde con el modelo: el proveedor de identidad y el broker aportan Identidad; los roles, los logins de sesión y el job de vencimiento aportan Grant; la regla de red detrás de la flecha punteada y el gateway aportan Ruta; el runner y el pipeline aportan Ejecución; el almacén de auditoría y los jobs de revisión aportan Registro. Una caja que no aporta ninguna parte es candidata a eliminarse, y la pregunta de Robin merece esa prueba.

El estado objetivo es una dirección y no una meta, así que aquí los pasos del libro se agrupan en cinco etapas. La agrupación es mía. Sigue cuánto puede cambiar un equipo antes de que el siguiente cambio se vuelva más fácil, y distintas organizaciones la reordenarán; el único orden que defendería es que el inventario va primero, porque sin él nadie sabe qué debe

cubrir cada paso posterior. Además, lo mejor es empezar pronto a llevar registros en una forma simple, ya que cada etapa posterior es más fácil de juzgar con evidencia.

Una etapa está terminada cuando su prueba de salida pasa, y “lo instalamos” no es una prueba. Cada prueba es una observación que alguien puede repetir.

Etapa	Qué añade	Prueba de salida	Por qué los equipos se estancan aquí
1. Conocer las puertas	Inventario de principals, puertas y copias (capítulos 1, 2, 13)	Un documento enumera cada puerta con un responsable y una fecha de fin, y un ingeniero nuevo puede responder “¿quién puede alcanzar el primary?” solo con él	El inventario se trata como un proyecto, así que nunca se termina y nunca se le da confianza
2. Nombrar a cada principal	Login y rol propios por principal, credenciales dinámicas, presupuestos por rol (capítulos 3, 4, 5)	Un día del log de la base de datos no contiene ningún login compartido, y ningún servicio se conecta como el propietario o el administrador	El cambio parece un riesgo de caída para cada servicio, así que la contraseña compartida se queda “por ahora”
3. Una ruta, acotada	Gateway, réplicas, JIT y break-glass, clasificación y enmascaramiento (capítulos 6, 7, 8, 12)	Una conexión directa al primary desde un laptop falla, un grant JIT desaparece en <code>expires_at</code> sin que nadie actúe y una sesión de lectura no devuelve ninguna columna <code>Restricted</code>	El miedo al gateway como punto único de fallo, y aprobadores que no pueden responder de noche
4. Mediada y registrada	Política de ejecución, runner, correcciones revisadas, migraciones solo por pipeline (capítulos 9, 10, 11)	Se rechaza una escritura desde el cliente propio de una persona, y el runner rechaza un conjunto de sentencias cuyo hash difiere de <code>approved_hash</code>	La revisión se vuelve el cuello de botella, así que las excepciones se acumulan más rápido que las funcionalidades de producto
5. Demostrable y ensayada	Almacén de auditoría con cadena de hashes, reglas de alerta, revisión de accesos, game days (capítulos 14, 15)	Una sentencia no anunciada elegida de ayer puede rastrearse hasta una persona, una solicitud y un resultado sin preguntar a nadie; <code>verifyChain</code> pasa; un game day termina con el acceso revocado confirmado desde el registro	El log existe y nadie lo lee, y el game day es lo primero que se re corta cuando un release se retrasa

Una etapa puede fallar su prueba después de darse por terminada, porque la configuración sufre drift; una prueba que se repite según un calendario muestra el drift mientras aún es pequeño.

Nota: Un equipo pequeño con una base de datos y cuatro ingenieros podría razonablemente detenerse en la etapa tres y dejar por escrito qué haría que continuara. La disciplina es la razón escrita, no la altura.

La lista de verificación

La lista de verificación está agrupada por pilar, y cada línea es una observación. La convención para usarla es la misma que para las pruebas de salida: un “no” es un hallazgo con un responsable y una fecha, no un fallo que se justifica con explicaciones.

```
function runChecklist(items):
  findings = []
  for each item in items:
    answer = observe(item.check)    // read a system, a log or a
    table, never ask what is intended
    if answer != "yes":
      findings.add({ item: item.id, owner: nominate(item), due:
        dateByWhichToDecide })
    // a finding may end in "accepted, with reason, until a date"; it
    may not end in silence
  return findings
```

Autorización

- ☐ Cada tabla pertenece a `ledgerly_owner`, y ninguna persona ni servicio tiene su login.
- ☐ Cada servicio se conecta con un login que es miembro de su propio rol `app_*`, y ningún rol de aplicación tiene DDL.
- ☐ No existe acceso de escritura permanente para ninguna persona; `human_readonly` es el valor por defecto y `human_jit_write` es un techo y no un rol concedido.
- ☐ Cada grant JIT y de break-glass tiene un `expires_at`, y el job de revocación y su comprobación independiente de vencidos se ejecutaron recientemente.

- ☐ Se rechaza una conexión directa al primary desde un laptop, y las réplicas rechazan el login de administrador.
- ☐ Las columnas Restricted no aparecen en ninguna solicitud JIT, ninguna sesión de lectura ni ninguna copia analítica, y el catálogo de clasificación enumera todas las columnas que existen.
- ☐ Cada backup, dump, export y copia de staging tiene un responsable con nombre y un vencimiento.

Ejecución

- ☐ Las sentencias de personas y de automatizaciones llegan a los datos a través de la política de ejecución, y la matriz deny / gate / allow está en control de versiones con un responsable.
- ☐ Las sesiones de escritura se ejecutan con auto-commit desactivado y un row cap, y los guarded settings no pueden cambiarse desde dentro de una sesión.
- ☐ Una escritura JIT nombra una corrección revisada, y el runner rechaza cualquier conjunto de sentencias cuyo hash difiera de `approved_hash`.
- ☐ Los cambios de vínculo con el propietario y las escrituras en columnas Restricted van a la ruta de producto en todos los lanes.
- ☐ Los cambios de esquema se ejecutan como `ledgerly_migrator` desde el pipeline, y ninguna persona tiene ese login.
- ☐ El paso de export se ejecuta con un rol estrecho, y su archivo tiene un responsable con nombre.

Rendición de cuentas

- ☐ Cada login se corresponde exactamente con una persona o una instancia de workload, y el log de la base de datos lo muestra.
- ☐ Cada escritura y cada registro de sesión elevada lleva la solicitud, el aprobador, el fingerprint y las filas devueltas, y los literales están con redaction.
- ☐ El almacén de auditoría está en una cuenta en la que los administradores de la base de datos y `ledgerly_access_admin` no pueden escribir, y `verifyChain` pasa.
- ☐ Un hueco en la secuencia, una sesión de gateway sin sesión de base de datos, o a la inversa, genera un hallazgo.

- ☐ `evaluateAlertRules` y `runAccessReview` se ejecutaron en el último ciclo, y las revocaciones que propusieron se aplicaron o se aceptaron con una razón.
- ☐ `containCompromise` se ha ensayado en un game day, y la consulta del radio de impacto se respondió desde el registro.

Cultura: el camino seguro tiene que ser el fácil

Todo control del libro es un costo para quien lo usa, y esa persona decide, en la peor noche de la semana, si lo paga. Un control lento o que cae a menudo se esquivo porque el diseño hizo que la ruta segura fuera la difícil, y un incidente no espera. La respuesta útil es tratar el esquivo como un defecto del diseño.

Pondría una regla en el centro de la cultura, y la haría medible:

```
function reviewBypass(signal):
    // signal: someone used a second door, a shared login, or an
    // unplanned break-glass
    cause = ask("what made the intended path unusable at that
    moment?")
    if cause in ["slow approval", "gateway unavailable", "missing
    product feature", "grant too narrow"]:
        // a design defect: repair the path, and note the date by
        // which it should be easy
        openFinding(cause, owner: "platform", due: nextReview)
    else if cause == "unknown":
        // an unexplained bypass stays open until it is understood
        keepOpen(signal)
    // nothing here assigns the signal to a person
    return cause
```

Dos hábitos respaldan la regla. Cuenta con qué frecuencia se declara el rol de break-glass, y pregunta cada vez si debería haber servido una ruta JIT o una funcionalidad de producto; un recuento al alza dice que la ruta normal es demasiado lenta, y un recuento de cero durante un año puede significar que el rol funciona o que nadie cree que se vaya a permitir, algo que resuelve un game day. Y da a los aprobadores algo que puedan responder de noche: una aprobación que tarda horas enseña a los solicitantes a pedir más de lo que necesitan, y una que siempre se concede en segundos enseña a nadie a leerla (el Capítulo 8 cubre la lista de verificación del aprobador).

Trade-offs y costos

Un capítulo de cierre que solo recomienda repetiría el error contra el que advierte el libro, así que aquí está lo que cuesta el diseño completo. El sistema de la arquitectura de referencia tiene muchas piezas, y cada una debe estar disponible en el momento en que alguien la necesita. El gateway, el runner, el pipeline, el broker, el almacén de auditoría y los jobs de revisión tienen cada uno un responsable, una ruta on-call y un modo de fallo. Un equipo de plataforma pequeño puede descubrir que dedica su tiempo a mantener los controles en marcha, y ese es un costo real que hay que poner frente a la exposición que elimina.

La lista de verificación y la escalera tienen costos propios. Una lista de verificación que se lee cada trimestre se convierte en un ritual a menos que se cierren los hallazgos, y una escalera se vuelve un informe de estado cuando los equipos optimizan para el nombre de la etapa; la prueba de salida es el antídoto, porque es barata de repetir y difícil de falsear. Una puntuación de “fuerte” dice para qué está construido un patrón, no que un despliegue concreto suyo lo cumpla.

Cuándo no usar el aparato de este capítulo: un sistema de un solo ingeniero con una base de datos y sin proveedores necesita un login de propietario que nadie use, un rol de solo lectura, un backup que se restaure y un inventario de una página. El libro es para el punto en que el número de principals o el costo de un error superan eso, y no sé dónde está ese punto en tu empresa; depende de cuántas personas pueden alcanzar los datos, cuántos de ellos son Restricted y cuánto tiempo permanece invisible un error.

Errores comunes

- **Tratar la lista de verificación como un certificado.** Es tentador terminar una pasada y archivar el resultado. La lista muestra dónde falta evidencia el día en que se leyó. Vuelve a ejecutarla según un calendario y mantén abiertos los hallazgos hasta que cada uno se resuelva o se acepte con una razón y una fecha.
- **Construir primero el patrón de más a la derecha.** Un ejecutor que media las sentencias resulta atractivo porque parece el que más control da. Sin el inventario y sin principals distintos debajo, media las

sentencias de un login compartido. Sube la escalera en orden, o deja por escrito por qué te conviene otro orden.

- **Añadir una segunda puerta para emergencias.** Parece resiliencia y reconstruye la contraseña compartida. Si un equipo necesita una mientras madura el gateway, necesita un responsable, una alerta al usarse y una condición de retiro (Capítulo 8).
- **Leer un esquivé como un problema de conducta.** Cuando alguien rodea un control, el diseño lo hizo más barato que la ruta prevista. Pregunta qué costaba la ruta prevista en ese momento, y cámbialo.

Ideas clave

- El acceso a bases de datos en producción es un problema de autorización, ejecución y rendición de cuentas, y una contraseña toca solo el primero de ellos, y solo en la puerta.
- El trabajo sobre las credenciales del Capítulo 4 es necesario porque la identidad hace posible el registro, y no es suficiente porque no dice nada sobre lo que un principal puede hacer ni sobre lo que hace una sentencia.
- Elige entre acceso directo, un bastión, un access gateway, un ejecutor que media las sentencias y una ruta de producto por tarea, con el patrón más barato que aún evidencie los pilares que la tarea pone en riesgo.
- Una etapa de madurez está terminada cuando su prueba de salida pasa, y una prueba que alguien puede repetir es la única forma de “terminado” que sobrevive al drift.
- Cada línea de la lista de verificación es una observación de un sistema, un log o una tabla; un “no” es un hallazgo con un responsable y una fecha, nunca un veredicto sobre una persona.
- El camino seguro tiene que ser el fácil, y cada esquivé es información sobre dónde el diseño volvió costosa la ruta prevista.
- Los controles reducen el radio de impacto y hacen visible el comportamiento, y cuestan disponibilidad, atención y espera, así que nombra esos costos antes de adoptarlos y de nuevo cuando se pagan.

Alex responde a la pregunta de Robin a la mañana siguiente con una prueba y no con una defensa: una pieza se queda si aporta una parte del modelo que nada más aporta y su prueba de salida pasa. El cuestionario volverá, de otro cliente o de otro auditor, y las mismas dos líneas estarán en él. Lo que el libro puede dejarte es un sistema que responde a la primera línea desde un documento, responde a la segunda desde un registro y permite que quien está on-call haga lo correcto sin un esfuerzo heroico.

Apéndice

Este apéndice es material de referencia para los tres pilares desde los que argumenta el libro: autorización, ejecución y rendición de cuentas. Contiene seis partes: un glosario, una referencia rápida de los patrones puntuados en el Capítulo 16, la matriz deny / gate / allow del Capítulo 9, una correspondencia entre las preguntas de los auditores y los artefactos, y dos plantillas, una para la revisión de accesos del Capítulo 14 y otra para la revisión posincidente del Capítulo 15.

Nada de lo que hay aquí es nuevo. Cuando una tabla repite una de un capítulo, las celdas son las del capítulo, y si las dos difieren, la fuente es el capítulo. Tus equipos jurídico y de cumplimiento son responsables de la interpretación de cualquier régimen; la correspondencia enumera artefactos de ingeniería y no afirma nada más.

A. Glosario

Los términos se enumeran alfabéticamente, con el capítulo que construye cada uno. Esta es la referencia final del libro para su vocabulario: cada entrada es la forma breve, y el capítulo indicado contiene la explicación completa.

Término	Significado
acceso just-in-time (JIT)	Acceso concedido para una tarea específica y un tiempo limitado (Capítulo 8).
acceso permanente	Acceso que existe de forma continua se use o no (Capítulo 1).
access gateway	Un intermediario consciente de la identidad que autentica a una persona, emite una credencial de corta duración y registra la sesión. No es un bastion ni un proxy de base de datos (Capítulo 7).

Término	Significado
access_audit_record	Las entradas que se escriben por cada sentencia, una de tipo intent y otra de tipo outcome, con el principal, el login, la solicitud, el <code>fingerprint</code> de la sentencia, las tablas y columnas tocadas y, en el outcome, los conteos de filas y el resultado. Es la parte Registro del modelo (Capítulo 14).
allowlist, denylist	Una lista de lo que se permite y una lista de lo que se rechaza. Nunca “whitelist” ni “blacklist” (Capítulo 7).
análisis de consultas	Inspeccionar una sentencia antes de que se ejecute: su tipo, las tablas y columnas que toca, si una escritura tiene una guarda de filas acotada y el conteo de filas esperado. Reduce errores y no es el límite de autorización (Capítulo 9).
approved hash	El <code>approved_hash</code> de una solicitud de acceso: el hash del conjunto de sentencias aprobado, tomado del registro de merge de la corrección revisada o del registro de la solicitud aprobada, nunca de la entrada del runner (capítulos 9, 10).
aprobación vinculada a la sentencia	Una aprobación ligada al conjunto exacto de sentencias que se ejecutará, identificado por <code>approved_hash</code> , en lugar de solo a una ventana de tiempo o a un conjunto de privilegios (capítulos 8, 9, 10).
bastion	Un host reforzado que es la única ruta de red hacia un sistema protegido. Reenvía el tráfico y no entiende el protocolo de la base de datos (Capítulo 7).
break-glass	Acceso de emergencia con logging intenso y revisión obligatoria. El rol es <code>human_breakglass</code> y la solicitud es autodeclarada (Capítulo 8).
cadena de hashes	Un formato de registro en el que cada entrada lleva el hash de la anterior, de modo que una entrada modificada o eliminada rompe <code>verifyChain</code> . Da evidencia de manipulación y no la impide (Capítulo 14).
cambio de vínculo con el propietario	Un <code>UPDATE</code> de una columna que vincula una fila con el cliente al que pertenece, como <code>customer_id</code> en <code>bank_accounts</code> . Es gate y se enruta a la ruta de producto (Capítulo 9).
clases de datos	Las cuatro clases de datos con nombre: Restricted (la exposición es difícil de deshacer, como <code>iban</code> y <code>token_value</code>), Confidential (datos personales o comerciales sobre clientes y contactos), Internal (datos operativos sin contenido personal), Public (seguros de publicar). Se escriben con mayúscula cuando se usan como nombres de clase (capítulos 2, 12).
clasificación de datos	Ordenar los datos en las clases con nombre que deciden las reglas de manejo (Capítulo 12).
consulta ad hoc	Una consulta que una persona escribe y ejecuta una vez, fuera de cualquier aplicación (Capítulo 10).

Término	Significado
contención	Detener el acceso de un principal en un orden fijo: revocar grants, cerrar sesiones, rotar credenciales, acotar el daño, registrar y notificar (Capítulo 15).
copia analítica	El almacén <code>ledgerly_analytics</code> . Es una copia de los datos de producción, no una réplica (Capítulo 6).
copia de datos	Cualquier lugar donde se hayan duplicado datos de producción: réplicas, la copia analítica, backups, dumps, exports, datos de staging (capítulos 6, 13).
corrección de datos	Un cambio en los datos de producción hecho por una persona fuera de la ruta normal de la aplicación, para reparar un estado que la aplicación no puede reparar (Capítulo 10).
credencial	Lo que un principal presenta para autenticarse: una contraseña, un certificado, un token. Una credencial es un tipo de secret (Capítulo 4).
credencial de corta duración	Una credencial que vence por sí sola en un tiempo breve y definido (Capítulo 4).
credencial dinámica	Una credencial de corta duración generada bajo demanda para un principal, de modo que dos solicitudes nunca reciben el mismo valor (Capítulo 4).
credencial estática	Una credencial que permanece igual hasta que alguien la cambia (Capítulo 4).
deny / gate / allow	Las respuestas de la política de ejecución, siempre en este orden. Deny nunca se ejecuta por esta ruta. Gate se ejecuta solo tras una aprobación adicional, o solo por la ruta de producto. Allow se ejecuta dentro de los grants de la sesión (Capítulo 9).
ejecución	Cómo llega una sentencia a los datos: quién o qué la envía, si algo la inspecciona antes de ejecutarse, qué acota su efecto y cómo se hace commit (capítulos 1, 9).
ejecución mediada	Un modo en el que el principal nunca tiene una conexión: envía una sentencia, el gateway o un runner la analiza y el runner la ejecuta (Capítulo 9).
enmascaramiento	Reemplazar u ocultar valores para que no se vea el valor real mientras la forma sigue siendo utilizable. El enmascaramiento estático cambia una copia, el dinámico cambia lo que devuelve una consulta (Capítulo 12).
four-eyes	Aprobación en la que una segunda persona revisa antes de que se ejecute una acción riesgosa. Dice quién aprueba, no qué se aprueba (Capítulo 8).
grant	Dar un privilegio a un rol, un rol a un login o a otro rol o, solo para JIT, un privilegio sobre columnas con nombre directamente a un login de sesión. También el registro resultante (Capítulo 3).
grant a nivel de columna	Un grant de un privilegio sobre columnas con nombre de una tabla. Donde el motor no lo admite, las columnas Restricted viven en tablas separadas fuera del techo, y los grants JIT bajan a nivel de tabla (capítulos 3, 8).

Término	Significado
guarded settings	Ajustes de sesión que un principal no puede cambiar desde dentro de una sesión: el timeout de sentencia, el row cap, el modo de auto-commit, el valor por defecto de solo lectura, el rol. Solo se llaman no sobrescribibles donde un mecanismo lo impone (capítulos 5, 9).
identidad	La afirmación verificada de que un principal es quien dice ser. Un rol no es una identidad (Capítulo 1).
independiente del motor	La expresión del libro para la notación y los consejos que no pertenecen a ningún motor de base de datos (Prefacio).
login	La cuenta a nivel de base de datos con la que se conecta un principal (capítulos 1, 3).
migración	Un cambio de esquema versionado que ejecuta <code>ledgerly_migrator</code> . “Cambio de esquema” es la idea general, “migración” el artefacto (Capítulo 11).
minimización	No recopilar, copiar ni conservar datos que nadie necesita (Capítulo 12).
mínimo privilegio	Conceder solo lo que una tarea necesita, durante el tiempo que lo necesita, contando el costo de hacerlo (capítulos 1, 3).
PII	Información de identificación personal. “Datos sensibles” es el término general para los datos Restricted y Confidential (Capítulo 2).
primary, réplica	La base de datos escribible y sus copias de lectura. La copia analítica no es una réplica (Capítulo 6).
principal	Cualquier cosa que se autentica: una persona, un servicio, un job, una integración de un proveedor (Capítulo 1).
principal de automatización	Un principal que no es una persona: un job, un runbook, un comando de ChatOps, un agente de IA. Tiene una workload identity y un valor por defecto de solo lectura, y el registro nombra tanto a él como a la persona que lo activó (Capítulo 9).
privilegio	Una cosa específica que permite un grant, como <code>SELECT</code> sobre una tabla (Capítulo 1).
proxy de base de datos	Un intermediario consciente del protocolo que agrupa, enruta o filtra conexiones de base de datos. Se llama pooler cuando el tema es solo el pooling (capítulos 5, 7).
puerta lateral	Una copia de datos o una ruta de export que esquiva los controles de acceso del primary (capítulos 6, 13).
radio de impacto	Cuánto puede verse afectado si un principal, una credencial o una acción sale mal. Siempre se enuncia sobre algo concreto (capítulos 1, 2).
redaction	Eliminar por completo un valor de una salida o de un log (Capítulo 12).

Término	Significado
registro	La parte del modelo que pregunta qué evidencia queda y si une la sentencia con la persona, la solicitud y el resultado (Capítulo 14).
registro de auditoría	El registro de quién hizo qué a qué datos, guardado donde quien actuó no puede modificarlo. La tabla <code>audit_events</code> es el registro de negocio de la aplicación, no el registro de auditoría (Capítulo 14).
revisión de accesos	Una comprobación programada de que el acceso de cada principal sigue siendo necesario. Compara lo que está concedido con lo que se ha usado, y decide un revisor con nombre (Capítulo 14).
revocación, rotación	La revocación hace que una credencial o un grant deje de funcionar ahora. La rotación reemplaza una credencial según un calendario o ante una sospecha, y no cierra las sesiones que ya están abiertas (capítulos 4, 15).
rol	Un conjunto de privilegios con nombre en la base de datos. Un rol nunca es una persona (Capítulo 3).
runbook	Un procedimiento escrito y seguido para una tarea repetible (Capítulo 10).
runner	El runner de correcciones de datos revisado. Ejecuta un conjunto de sentencias solo si su hash es igual a <code>approved_hash</code> , y el gateway reenvía las escrituras JIT solo desde él (Capítulo 10).
ruta de producto	Una operación definida en una herramienta, como <code>admin-console</code> pidiendo a <code>payments-service</code> que cambie una columna Restricted, sin SQL para la persona (capítulos 9, 10).
secret	Cualquier valor que concede acceso si se revela (Capítulo 4).
sesión	Una conexión autenticada o una interacción mediada por el gateway de un principal (capítulos 5, 7).
session handout	Un modo en el que el gateway da a un principal un login de sesión y una conexión, y el cliente propio del principal envía las sentencias. Los grants del login acotan todo (Capítulo 9).
teatro de seguridad	Un control que añade fricción sin reducir la exposición (capítulos 2, 8).
techo	Lo máximo que puede pedir una solicitud JIT, definido por <code>human_jit_write</code> . Nunca una columna Restricted (capítulos 3, 8).
tokenización	Reemplazar un valor por un sustituto que solo un servicio controlado puede volver a mapear. No es ni cifrado ni enmascaramiento (Capítulo 12).
workload identity	Una identidad que la plataforma da a un workload en ejecución para que pueda autenticarse sin una contraseña almacenada (Capítulo 4).

Varios de estos pares se confunden, y una regla breve resuelve cada uno antes de que empiece una discusión.

Si la pregunta es	Usa	No
Un host que reenvía tráfico de red, sin conocimiento de las sentencias	bastion	access gateway
Un intermediario que emite logins de sesión por persona y registra las sesiones	access gateway	proxy de base de datos
Agrupar o enrutar conexiones, nada sobre las personas	pooler, o proxy de base de datos	access gateway
El cliente propio de la persona envía sentencias por una conexión que el gateway le dio	session handout	ejecución mediada
La persona envía una sentencia y nunca tiene una conexión	ejecución mediada	session handout
Impedir que una credencial siga funcionando ahora	revocación	rotación
Quién aprueba una acción riesgosa	four-eyes	aprobación vinculada a la sentencia
Qué se aprobó exactamente, por hash	aprobación vinculada a la sentencia	four-eyes
El registro de negocio de la aplicación	audit_events	registro de auditoría

B. Referencia rápida de patrones

El Capítulo 16 puntúa cinco patrones frente a los tres pilares. Las puntuaciones son el criterio del libro y no una medición: “fuerte” significa que el patrón está construido para demostrar el pilar, “parcial” que demuestra una parte o solo con algo añadido, “ninguna” que guarda silencio. La columna de costo es la que da el Capítulo 16, abreviada.

Patrón	Qué resuelve	Costo	Autorización	Ejecución	Rendición de cuentas	Capítulos
Acceso directo (del cliente a la base de datos)	Existe un login y sus grants eran los que eran	Ninguno para construirlo; todos los costos llegan después, como una semana de rastreo	Parcial: solo grants, el alcance es el que permita la red	Ninguna	Parcial en el mejor de los casos: logins por persona si alguien los provisiona, de lo contrario ninguna	1

Patrón	Qué resuelve	Costo	Autorización	Ejecución	Rendición de cuentas	Capítulos
Bastion	Un host controlado, un solo lugar donde cortar la ruta de red	Un host que parchear y mantener disponible	Parcial: reduce la ruta de red	Ninguna: reenvía, no distingue un <code>SELECT</code> de un <code>DROP</code>	Parcial: quién inició sesión en el host, no qué se ejecutó	7
Session proxy o access gateway	Una persona con nombre tuvo un login de sesión específico, entre dos marcas de tiempo, bajo grants específicos	Un sistema nuevo que debe estar disponible cuando alguien lo necesita, y la ruta de aprobación que lo acompaña	Fuerte: logs por sesión, vencimiento, grants por solicitud	Parcial: acotada por los grants y los guarded settings, las sentencias se escriben libremente	Fuerte en quién y cuándo, parcial en qué	7, 8
Ejecutor que media las sentencias	Este conjunto exacto de sentencias, por hash, fue aprobado, inspeccionado, acotado y contado	Cada tarea espera una revisión, nada es interactivo, alguien construye y mantiene el runner	Fuerte	Fuerte: se inspecciona antes de ejecutarse, vinculada a un approved hash	Fuerte: sentencia, aprobador y resultado en un solo registro	9, 10
Ruta de producto (sin SQL)	Se ejecutó una operación definida, para un rol en una herramienta	Una funcionalidad por tarea, y el ciclo de cambio más lento de los cinco	Fuerte: la operación es el permiso	Fuerte: la operación es fija	Parcial: la persona queda en el rastro de auditoría de la herramienta, el log de la base de datos muestra el login del servicio	9, 10

Elegir entre ellos es una decisión por tarea, y la regla es la que da el Capítulo 16: el patrón más barato que aún evidencie los pilares que la tarea pone en riesgo.

```
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
    (Chapter 11)
    if task.changesSchema:
        return "migration pipeline"
    // Restricted columns and owner links are changed by the product,
    not by SQL
    if task.writesRestrictedColumn or task.changesOwnerLink:
        return "product path"
    if task.writesData:
        // reviewed statement set, approved hash, runner executes
        return "statement-mediating executor"
    if task.readsData:
        return "access gateway: human_readonly on ledgerly_replica_2"
    // anything else is an exception with a named owner and an end
    return "exception: owner, reason, ends_on"
```

C. Matriz de operaciones peligrosas

La matriz siguiente es la política de ejecución del Capítulo 9, repetida con sus celdas sin cambios. Las filas son las clases de sentencia que produce el clasificador. Las columnas son los lanes por los que puede llegar una sentencia. Cada celda tiene una de tres respuestas: deny (nunca se ejecuta por esta ruta), gate (necesita una aprobación que aún no tiene, o pertenece a otra ruta), allow (se ejecuta dentro de los límites del bucle, y donde una celda dice “bound”, solo dentro del límite de la fuente nombrada).

Clase de sentencia	Sesión de lectura humana	Escritura JIT	Break-glass	Runner de correcciones de datos	Migración
Lectura, dentro del row cap	Allow	Allow	Allow	Allow	Allow
Escritura acotada, dentro del conteo aprobado	Deny	Allow, bound a la corrección (solo runner)	Allow, bound a un row cap (nota obligatoria)	Allow, bound al approved hash y al conteo	Allow, bound a la regla de backfill
Escritura sin acotar	Deny	Deny	Deny	Deny	Gate: aprobador designado

Clase de sentencia	Sesión de lectura humana	Escritura JIT	Break-glass	Runner de correcciones de datos	Migración
Cambio de vínculo con el propietario	Deny	Gate: ruta de producto	Gate: ruta de producto	Gate: ruta de producto	Gate: aprobador designado
Escritura en una columna Restricted	Deny	Gate: ruta de producto	Gate: ruta de producto	Gate: ruta de producto	Gate: aprobador designado
Export	Gate: por nivel	Deny	Deny	Deny	Gate: aprobador designado
Cambio de esquema (DDL)	Deny	Deny	Deny	Deny	Allow: lane revisado
Cambio de privilegios	Deny	Deny	Deny	Deny	Gate: revisión de plataforma
Escritura en <code>ledger_entries</code> o <code>audit_events</code>	Deny	Deny	Deny	Deny	Gate: revisión de plataforma

Tres reglas de lectura ahorran la mayoría de las consultas.

- **Una escritura en manos de una persona se deniega o se acota.** La sesión de lectura humana no tiene ruta de escritura, las escrituras JIT vienen solo del runner, y el break-glass es el único lane en el que una persona escribe directamente, bajo un row cap y con una nota.
- **Un gate nombra adónde va después la sentencia.** “Ruta de producto” significa que `admin-console` se lo pide a `payments-service`, una segunda persona aprueba y el cambio queda registrado en `audit_events`. “Aprobador designado” y “revisión de plataforma” significan que el lane de migración marca la sentencia para una persona que tiene que decir que sí.
- **La matriz no es el límite de autorización.** El análisis de consultas reduce errores. Los grants del login siguen siendo el límite, y una sentencia que el clasificador interpreta mal sigue limitada por ellos (Capítulo 9).

La decisión para una sentencia, escrita como regla:

```
function answerFor(statementClass, lane):
  // the matrix above is the only source; no lane has its own rule
  set
    cell = matrix[statementClass][lane]
    if cell starts with "Deny":
      refuse and record the refusal
    else if cell starts with "Gate":
      // the text after the colon names the route or the approver
      hold the statement; route it as the cell says; do not run it
  as sent
    else:
      // "bound" cells compare the real count with the bound before
  COMMIT
    run within the loop's limits
```

D. Correspondencia con requisitos de cumplimiento

Los regímenes de protección de datos y de tarjetas de pago suelen plantear una familia de preguntas sobre el acceso. Un auditor que trabaja bajo uno de estos regímenes querrá ver normalmente que un control existe, que se ejecutó y que se miraron sus resultados. Esta correspondencia enumera los tipos de pregunta, el control que responde a cada una, el artefacto que deja y el capítulo que lo construye. Es la tabla del Capítulo 14, repetida, y no afirma que ningún control satisfaga ninguna obligación.

Pregunta	Control	Artefacto	Ca- pí- tulo
¿Quién puede al- canzar los datos de producción y por qué ruta?	El gateway como única puerta, es- quema de roles como código, comprobación de drift	Lista de reglas de red, definiciones de roles, informe de drift	3, 7
¿Quién ejecutó una sentencia dada, con qué grants y qué aprobación?	access_audit_record con id de solicitud y aprobador	El registro y el informe de verifyChain	14
¿Cuánto dura el ac- ceso elevado y quién lo aprobó?	Solicitud JIT con aprobador y expires_at, job de revocación	Registros de solicitu- des, log de revocacio- nes, comprobación de grants vencidos	8

Pregunta	Control	Artefacto	Ca- pí- tulo
¿Cómo se identi- can y se limitan los datos sensibles?	Clasificación en el esquema, en- mascaramiento, columnas retenidas	El catálogo, la verifica- ción de la carga enmascarada	12
¿Dónde están las copias de los datos de producción y quién es responsa- ble de cada una?	Registro con responsable y fecha de fin, sweeper	Registro, log del sweeper	6, 13
¿Quién cambia el esquema y quién lo revisó?	Pipeline de migraciones, rama protegida, entrada de inicio externa	Registros de migracio- nes cotejados con com- mits y entradas de auditoría	11
¿Sigue siendo ne- cesario el acceso?	Revisión de accesos con revisores con nombre	Revisión completada con decisiones	14
¿Qué ocurre en una emergencia?	Break-glass con alerta y revisión obligatoria	Registros de revisión, historial de alertas	8, 15
¿Quién administra el propio acceso?	Logins de <code>ledgerly_access_admin</code> como productores separados	Sus flujos y la compa- ración entre el catálogo y el registro	3, 14

Un control que existe pero nunca se ejecutó, o se ejecutó y nadie lo miró, responde solo a un tercio de la pregunta. La comprobación siguiente es la que conviene ejecutar antes de entregar un artefacto a quien lo pidió.

```
function evidenceStatus(row):
  // three things an auditor tends to want, from Chapter 14
  if row.artifact is null:
    return "control not evidenced: build or find the artifact"
  if row.artifact.lastProducedAt is null:
    return "control exists, no evidence it ran"
  if row.artifact.lastReviewedBy is null:
    return "control ran, results not looked at"
  return "evidenced: attach artifact and its review"
```

E. Plantilla de revisión de accesos

Esta es la plantilla detrás de `runAccessReview` del Capítulo 14. Cópiala para cada ciclo de revisión. La revisión propone y decide un revisor con nombre; nadie revisa su propio acceso, y el período es más largo que el ci-

clo más largo del workload que se revisa (una revisión de un mes propondría revocar un grant que un informe trimestral usa en dos de cada tres meses).

Encabezado

Review cycle:	<period start> to <period end>
Reviewer(s):	<names; none is the owner of an item they review>
Record queried on:	<date>, verifyChain result: <passed failed: details>
Period chosen and why:	<longest workload cycle covered?>

Parte 1: grants declarados frente al uso

Grant (rol, privilegio, tabla) o login	Responsable	Último uso en el registro	Hallazgo	Decisión	Decidido por	Hecho el
<role>, <privilege>, <table>	<named principal>	<date or none>	sin usar / sin responsable / uso más estrecho que el grant / correcto	mantener / reducir a <privilege> / revocar / asignar responsable	<reviewer>	<date>

Coteja el uso por privilegio y no solo por tabla. Un rol que hace select todas las noches y nunca ha hecho update es candidato a reducirse de S, U a S. Incluye las asignaciones de grupos del proveedor de identidad a roles y los logins de ledgerly_access_admin.

Parte 2: lo que no son grants

Elemento	Pregunta	Respuesta	Responsable	Decisión
Registro de copias de datos (Capítulo 13)	¿Cada copia sigue teniendo un responsable y una fecha de fin futura?	<yes / no, which>	<name>	<keep / end by date>
Quién puede aprobar solicitudes JIT	¿Cada aprobador sigue en un rol que lo justifica?	<yes / no, which>	<name>	<keep / remove>
Quién puede leer el almacén de auditoría	¿Cada lector sigue necesítándolo?	<yes / no, which>	<name>	<keep / remove>

Elemento	Pregunta	Respuesta	Responsable	Decisión
Principals de automatización (jobs, runbooks, agentes de IA)	¿Cada uno sigue ejecutándose, con una workload identity y un valor por defecto de solo lectura?	<yes / no, which>	<name>	<keep / retire>

Parte 3: usos de break-glass en el período

Id de solicitud	Usado por	Incidente o razón	Revisado por	Fecha de revisión	Hallazgo sobre el sistema
<id>	<principal>	<reason>	<other than user>	<date or MISSING>	<for example: missing path>

Una fila sin fecha de revisión es en sí misma un hallazgo.

Reglas de cierre

```
function closeAccessReview(review):
  for each row in review.rows:
    // a row without a decision is not a review, it is a form
    if row.decision is null or row.decidedBy is null:
      raise ReviewIncomplete
    if row.decidedBy == row.owner:
      raise ReviewerIsOwner
  // decisions become tracked work with an owner and an end date
  for each row in review.rows where row.decision != "keep":
    createChange(row, owner: row.owner, endsOn:
review.nextCycleStart)
markClosed(review.id)
```

F. Plantilla de revisión posincidente

Esta plantilla sigue el Capítulo 15. Es sin culpas: cada pregunta trata del sistema que permitió una acción o la hizo invisible, y ninguna pregunta quién debería haberlo sabido. Complétala a partir del registro, no de memoria. Donde el registro no pueda responder una pregunta, escríbelo, porque el hueco es un hallazgo. Los tiempos son los que muestra el registro; no los estimes.

1. Resumen

Incident id: <id carried in every request reason>
 Opened / closed: <timestamps from the incident record>
 Trigger: <alert, report or declared exposure>
 Principal(s) in scope: <logins, services, automation principals>
 Pillars touched: <authorization / execution / accountability>

2. Contención, en orden

Paso	Hecho a las	Resultado	Notas sobre lo que se interpuso
Revocar grants	<timestamp>	hecho / fallido	
Cerrar sesiones	<timestamp>	hecho / fallido	
Rotar credenciales	<timestamp>	hecho / fallido	
Acotar el daño desde el registro de auditoría	<timestamp>	hecho / fallido	
Registrar y notificar	<timestamp>	hecho / fallido	

Preguntas: ¿Se ejecutó cada paso aunque uno anterior fallara? ¿Quedó algún paso sin terminar, y se le dio seguimiento? ¿Era accesible el log y, si no, las entradas quedaron en cola sin retrasar la retirada del acceso?

3. Radio de impacto

Query run: <everything the principal touched since T, Restricted rows per table>
 Tables touched: <list, with class>
 Restricted columns: <touched or not, rows returned per table>
 Agreement check: <does the scope agree with the register of copies and the gateway session list?>

4. Ciclo de vida del acceso durante el incidente

Id de solicitud	Tipo	Solicita-do	Concedi-do	Primer uso	Revoca-do	Revisado por
<id>	jit / breakglass	<time>	<time>	<time or none>	<time>	<other than user, or MISSING>

Preguntas: ¿Algún grant permaneció más allá de `expires_at`? ¿Se concedió algún acceso que ninguna solicitud explica? ¿`closeIncident` encontró acceso abierto, y se cerró antes de que se cerrara el incidente?

5. Escrituras durante el incidente

Escritura	Ruta (runner con revisión expedita, ruta de producto, break-glass directo)	Revisor que no fue el autor	El approved hash coincidió
<description>	<route>	<name>	sí / no

Pregunta: ¿Se usó el break-glass directo para una escritura donde habría servido el runner, y por qué?

6. Casos diferidos, si ocurrieron

- **Gateway no disponible:** cuánto tiempo se detuvo el acceso humano, si los servicios siguieron funcionando y si el runbook de restauración funcionó tal como estaba escrito.
- **Restauración durante el incidente:** si la copia restaurada se registró con un responsable y una fecha de fin, y si terminó.
- **Aprobaciones lentas por la noche:** qué solicitud esperó, a quién, y qué hizo la regla para una solicitud sin respuesta.

7. Hallazgos y seguimiento

Hallazgo sobre el sistema	Parte (Identidad, Grant, Ruta, Ejecución, Registro)	Cambio	Responsable	Termina el
<for example: a path that should have existed>	<part>	<change>	<name>	<date>

Cierra con una comprobación: ¿cada hallazgo es un cambio en un diseño con un responsable y una fecha de fin, y no un recordatorio de ser más cuidadoso?

SOBRE EL LIBRO

El acceso a bases de datos en producción no es un problema de distribución de credenciales. Es un problema de autorización, ejecución y rendición de cuentas.

Un gestor de contraseñas mejor solo cambia la forma en que un secret llega a una persona. No decide qué puede hacer esa persona, qué se ejecuta en realidad contra los datos ni quién responde por ello después. Este libro sigue a Ledgerly desde una contraseña de administrador compartida hasta un modelo de acceso diseñado a propósito: identidades de mínimo privilegio, acceso just-in-time y break-glass, ejecución mediada para las operaciones peligrosas y un registro de auditoría capaz de decir quién hizo qué y por qué. Los patrones son independientes del motor, y cada uno se presenta con el costo que conlleva.

SOBRE EL AUTOR

Muhammet Şafak desarrolla software desde 2008. Durante muchos años ha desempeñado funciones activas en los sectores de la salud, las finanzas y las telecomunicaciones, y sostiene que el desarrollo debe centrarse en el problema, no en la tecnología. Desde 2024 diseña y desarrolla sistemas de alto tráfico escalables de extremo a extremo.

Brinda mentoría a equipos de desarrollo de software pequeños y medianos, dirige proyectos de código abierto y deja por escrito sus decisiones, con sus razones y su costo.

Los sistemas confiables no se construyen con piezas confiables. Se construyen alrededor de las piezas que fallan.

I
ACCESO

II
EVENTOS

III
AGENTES