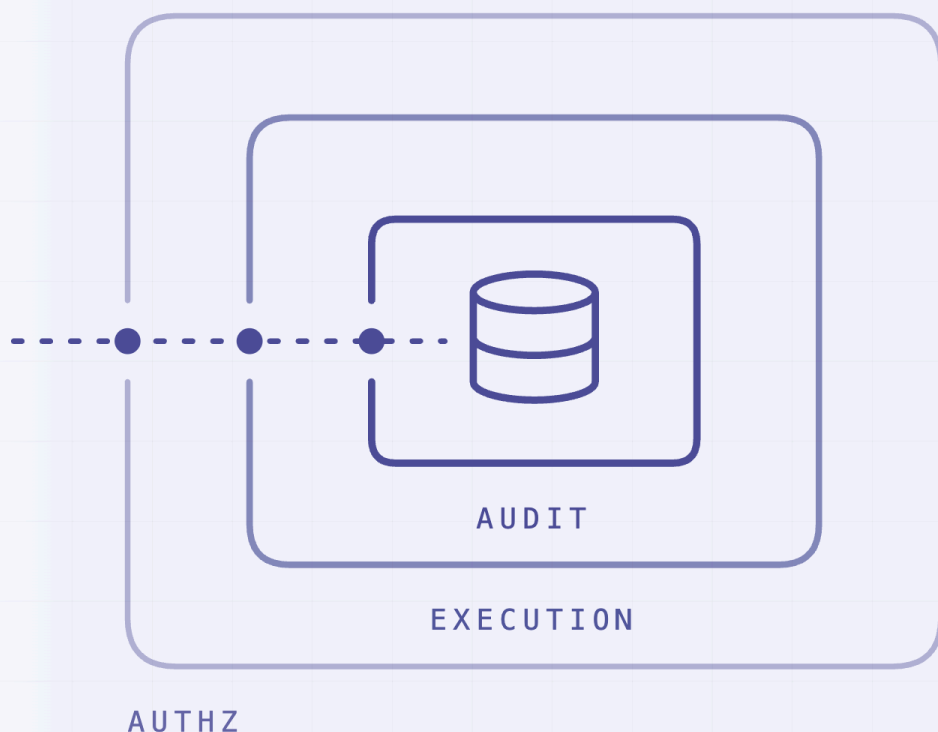




Guia de acesso a bancos de dados em produção

Guia de acesso a bancos de dados em produção

Identities, credenciais, acesso just-in-time, execução mediada e auditoria

Muhammet Şafak

Sumário

Prefácio	6
PARTE I — FUNDAMENTOS	13
Capítulo 1 — A porta que ninguém projetou	14
Capítulo 2 — Um modelo de ameaças para os dados de produção	29
Capítulo 3 — Identidades, papéis e privilégio mínimo	48
PARTE II — ACESSO DAS APLICAÇÕES	71
Capítulo 4 — Credenciais e secrets	72
Capítulo 5 — Conexões como política	91
Capítulo 6 — Leituras, réplicas e cópias de dados	110
PARTE III — ACESSO HUMANO	124
Capítulo 7 — Caminhos de entrada: bastions, proxies e access gateways	125
Capítulo 8 — Acesso just-in-time e break-glass	148
Capítulo 9 — Execução mediada: análise de consultas e operações perigosas	174
Capítulo 10 — Consultas ad hoc e correções de dados	190
Capítulo 11 — Mudanças de esquema e acesso para migrações	216
PARTE IV — PROTEGER OS DADOS	238
Capítulo 12 — Dados sensíveis: classificação, mascaramento e minimização	239
Capítulo 13 — Backups, exports e outras portas laterais	261
PARTE V — COMPROVAR E OPERAR	285
Capítulo 14 — Auditoria e responsabilização	286
Capítulo 15 — O acesso durante incidentes	305

Capítulo 16 — Um checklist de acesso e reflexões finais	323
Apêndice	338

Prefácio

Imagine uma quinta-feira à noite na Ledgerly, uma empresa de faturamento e pagamentos que não existe. Robin está on-call. A fatura de um cliente está presa no estado “sending” desde a tarde, e a correção parece pequena: uma linha, uma coluna de status. O cliente de banco de dados instalado no desktop da equipe lembra a senha de administrador compartilhada, conecta-se ao primary e faz commit automaticamente, como esses clientes costumam fazer. Uma correção de uma linha passa por esse cliente, sob um papel que pode alterar todas as tabelas, e a cláusula `WHERE` dela atinge um lote inteiro de faturas, em vez de uma só.

Robin passa o resto da noite no reparo. O que demora mais é a pergunta que vem depois: quem executou aquela instrução? O log do banco de dados mostra o papel de administrador compartilhado, que todos os serviços e, digamos, vinte pessoas usam. O gerenciador de senhas mostra quem abriu a entrada, não quem executou o quê. A equipe consegue reconstruir a alteração a partir dos efeitos colaterais na aplicação, e no fim consegue. Mas, por cerca de um dia, ninguém sabe dizer com segurança o que exatamente aconteceu, em quais linhas e se mais alguém usou a mesma porta ao mesmo tempo.

Nada nessa história foi irracional por si só. A senha foi compartilhada porque compartilhá-la era a forma mais rápida de colocar o primeiro serviço no ar. O cliente fazia commit automático porque esse é o comportamento padrão dele. O papel podia alterar todas as tabelas porque, no começo, era mais fácil do que decidir de quais tabelas cada serviço precisava. Cada passo foi razoável no dia em que foi dado. Juntos, construíram uma porta que ninguém projetou.

Essa porta é o assunto deste livro.

Por que este livro existe

O acesso a um banco de dados em produção costuma ser tratado como encantamento. Ele é configurado quando o primeiro serviço é implantado e depois só volta a ser tocado quando algo quebra ou alguém sai da empresa. Enquanto isso, o banco de dados se torna a coisa mais valiosa que a empresa opera: os clientes, o dinheiro, o histórico. A distância entre o quanto a empresa depende dele e o pouco design que se dedica ao acesso a ele é o assunto deste livro.

Muito já foi escrito sobre cada peça. A documentação de bancos de dados explica como criar papéis e grants. As ferramentas de secrets explicam como emitir credenciais. As orientações de segurança explicam por que o acesso permanente é um risco. O que é mais difícil de encontrar é o quadro completo do ponto de vista de quem faz engenharia: como as peças se encaixam, quanto cada uma custa a quem constrói e opera o sistema e em que ordem uma equipe com pouco tempo poderia adotá-las.

Este livro tenta ser esse quadro, e parte de uma única afirmação. O acesso a bancos de dados em produção não é um problema de distribuição de credenciais; é um problema de autorização, execução e responsabilização.

A visão padrão trata o acesso como uma questão de quem tem uma senha e de como ela é armazenada, rotacionada e distribuída. A quinta-feira à noite descrita acima mostra onde essa visão termina. Robin tinha uma credencial legítima, e a credencial fez exatamente o que foi emitida para fazer. O que falhou foi tudo ao redor dela: nada decidiu que essa pessoa deveria executar uma escrita naquela tabela, nada inspecionou a instrução nem limitou o efeito dela antes do commit, e nada vinculou o resultado a uma pessoa depois. A autorização decide o que pode acontecer e onde. A execução é a forma como cada instrução chega aos dados. A responsabilização é a evidência que permanece. Cada capítulo nomeia o pilar a que serve, e a Parte I constrói o vocabulário para os três. O livro trata cada controle como uma decisão com preço. Um controle adequado para a tabela de tokens de pagamento pode ser inútil para uma tabela de modelos de fatura, e o livro dirá isso quando for relevante.

Convém ser claro quanto à postura. O livro é escrito do lado de quem defende. As ameaças são descritas apenas na medida em que explicam por que um controle existe; nada aqui é um guia para invadir. Ele não vende

medo e não cita estatísticas de vazamentos, porque uma revisão de design calma é mais útil do que uma alarmante. E ele contabiliza o atrito com honestidade. Todo controle exige algo dos engenheiros e de quem está on-call, e um controle que acrescenta atrito sem reduzir a exposição é teatro de segurança. Quando achar que uma prática popular é teatro, direi isso e explicarei por quê.

Para quem é este livro

O livro é escrito para **engenheiros de backend e de plataforma** que constroem serviços que falam com um banco de dados em produção, para **papéis de SRE e DBA** que operam esse banco, para **tech leads** que precisam decidir como o acesso deve funcionar e para **engenheiros que passaram a ser responsáveis por segurança** em uma equipe que nunca teve uma função dedicada a isso. Se você já se perguntou se a senha compartilhada ainda é a mesma senha compartilhada, hesitou antes de conceder acesso de escrita a um colega ou ouviu a pergunta “quem pode ver esta tabela?” e percebeu que a resposta honesta exige um dia de investigação, você é o leitor que tenho em mente.

Você não precisa de formação em segurança. Convém estar à vontade com algumas coisas:

- Bancos de dados relacionais: tabelas, papéis e grants, transações e o que é uma réplica.
- Serviços que se conectam a um banco de dados por uma rede, e o vocabulário de connection pools e timeouts.
- O trabalho on-call: receber pages, consertar algo sob pressão e registrar por escrito, depois, o que aconteceu.

Conhecer um motor de banco de dados específico ajuda nos exemplos, mas não é obrigatório, como explica uma seção mais adiante.

O que este livro não é

Não é um livro de administração de bancos de dados. Ele não vai ensinar você a ajustar um planner, dimensionar um buffer ou escolher uma topologia de replicação. Onde a administração importa para o acesso, como no

caso das réplicas e das migrações, o livro cobre essa parte e deixa o resto para a documentação do próprio banco de dados.

Não é um manual de conformidade. Regimes como GDPR, KVKK, PCI DSS, SOC 2 e HIPAA aparecem porque moldam o que auditores e clientes perguntam, e o livro os descreve apenas nesse espírito: regimes desse tipo tendem a perguntar quem pode ver o quê, como o acesso é aprovado e que registro existe depois. Ele não interpreta nenhum regime, não cita seus artigos nem diz o que se exige de você. A interpretação cabe às pessoas de jurídico e conformidade, e o objetivo do livro é dar a elas uma arquitetura sobre a qual possam raciocinar.

E não é um livro de ataque. Você não encontrará técnicas para invadir. Quando um capítulo precisa de uma ameaça, ele nomeia o risco em uma frase, como uma credencial que vaza, uma consulta que cai no banco de dados errado ou uma sessão de fornecedor deixada aberta, e passa ao controle. Se descrever a ameaça com mais precisão não mudaria o design, o detalhe fica de fora.

Como o livro está organizado

O livro tem cinco partes e dezesseis capítulos, seguidos de um apêndice.

Parte I, Fundamentos (capítulos 1 a 3) defende a tese acima e constrói o vocabulário para o resto. O Capítulo 1 examina como as portas de um banco de dados se acumulam. O Capítulo 2 constrói um modelo de ameaças para os dados de produção que inclui tanto erros quanto uso indevido. O Capítulo 3 trata de identidades, papéis e privilégio mínimo, dos quais todos os capítulos seguintes dependem.

Parte II, Acesso das aplicações (capítulos 4 a 6) trata dos serviços. O Capítulo 4 aborda credenciais e secrets, das senhas estáticas às credenciais dinâmicas e à workload identity. O Capítulo 5 trata as conexões como política: pooling, limites e timeouts. O Capítulo 6 aborda leituras, réplicas e as outras cópias de dados que a produção cria discretamente.

Parte III, Acesso humano (capítulos 7 a 11) trata das pessoas. O Capítulo 7 compara bastions, proxies e access gateways. O Capítulo 8 aborda o acesso just-in-time e break-glass. O Capítulo 9 aborda a própria execução:

como um gateway inspeciona cada instrução, limita seu efeito e faz o commit dela. O Capítulo 10 aborda consultas ad hoc e correções de dados. O Capítulo 11 aborda as mudanças de esquema e quem pode fazê-las.

Parte IV, Proteger os dados (capítulos 12 e 13) desloca o foco de quem pode se conectar para o que essa pessoa pode ver. O Capítulo 12 aborda classificação, mascaramento e minimização. O Capítulo 13 aborda backups, exports e outras portas laterais que contornam todos os controles construídos até então.

Parte V, Comprovar e operar (capítulos 14 a 16) aborda como uma equipe sabe que seus controles funcionam e os mantém funcionando. O Capítulo 14 aborda logs de auditoria e revisões de acessos. O Capítulo 15 aborda o acesso durante incidentes, quando a pressão é maior e os atalhos são mais tentadores. O Capítulo 16 oferece um checklist, uma escada de maturidade com testes de saída e algumas reflexões finais sobre cultura.

O apêndice reúne um glossário, uma referência rápida dos padrões, uma matriz de operações perigosas, uma correspondência com requisitos de conformidade e modelos para revisões de acessos e revisões pós-incidente.

Todos os capítulos têm a mesma forma. Cada um abre com um cenário curto e abertamente fictício. Em seguida explica o problema e por que a abordagem óbvia fica aquém, apresenta o controle e expõe seus custos e quando não usá-lo. Lista os erros comuns, tratados como armadilhas do design e nunca como falhas pessoais, e termina com cinco a sete pontos-chave e uma ponte para o capítulo seguinte. A forma compartilhada permite que você leia os capítulos em ordem ou salte para o que corresponde ao problema desta semana, e ainda assim encontre a seção de custos onde espera.

Conheça a Ledgerly

A Ledgerly é um serviço fictício de faturamento e pagamentos entre empresas. As empresas emitem faturas por ela, recebem pagamentos e mantêm um ledger do que foi movimentado. Ela roda cinco serviços: `billing-api`, `invoice-worker`, `payments-service`, `reporting-job` e `admin-console`. O banco de dados `primary` tem duas réplicas e uma cópia de

analytics para a equipe de dados. Ele armazena dados de contato de clientes, informações de contas bancárias e tokens de pagamento, e é por isso que o design de acesso dela importa.

As pessoas ao redor dela cumprem papéis conhecidos. Há engenheiros de backend on-call, uma equipe de suporte, uma equipe de dados, um pequeno grupo de plataforma que opera os bancos de dados e um fornecedor externo de contabilidade que recebe exports.

No início do livro, a Ledgerly tem uma senha de administrador compartilhada, acesso permanente para a maior parte da equipe de backend, clientes desktop nos laptops e nenhum registro de quem fez o quê. No fim, tem um papel para cada serviço, credenciais de curta duração, um gateway para as pessoas, uma forma revisada de corrigir dados, instruções verificadas e limitadas antes de rodar, cópias mascaradas e um log que não pode ser reescrito em silêncio. Ela chega lá um passo de cada vez, e cada capítulo diz que passo acrescenta e quanto esse passo custa à equipe.

A Ledgerly é inventada, e tudo o que acontece com ela também. Cada capítulo abre com um cenário marcado como ficção, e qualquer número nele é hipotético. Nomes como Robin, Sam, Jordan e Alex são marcadores de posição para papéis em uma equipe. Quando um cenário dá errado, a lição trata do design que deixou isso acontecer, e nunca da pessoa na história.

Como ler o código

Nenhum exemplo deste livro é escrito em uma linguagem de programação real. Os procedimentos são escritos em pseudocódigo simples, e os papéis, grants e tabelas usam uma notação semelhante a SQL que não pertence a nenhum motor em particular. Nada no livro foi feito para ser colado em um terminal. Você não encontrará sintaxe de configuração de um proxy, de um vault ou de um produto de nuvem específico, porque essa sintaxe muda e o padrão por trás dela não.

Os diagramas usam Mermaid. Cada um é precedido por uma frase que diz o que observar, e os caminhos negados ou com falha são rotulados. Os exemplos nunca contêm secrets, números de conta ou endereços de e-mail realistas; marcadores como `<secret>` e `IBAN-PLACEHOLDER` ocupam o lugar deles.

Por que em inglês e por que independente do motor

O livro é escrito em inglês porque a maior parte do vocabulário desta área, de least privilege a break-glass, é usada em inglês mesmo em equipes que trabalham em outros idiomas, e queria que os termos continuassem reconhecíveis. O livro é independente do motor por um motivo relacionado. Uma equipe pode rodar PostgreSQL, MySQL ou outra coisa, e pode usar um serviço de identidade na nuvem, um gerenciador de secrets ou um script feito à mão. Os padrões são os mesmos em todos eles: separar o proprietário dos usuários, dar a cada workload a própria identidade, fazer as credenciais expirarem, colocar uma entrada que reconhece a identidade na frente das pessoas, verificar o que vai rodar antes de rodar, executar as escritas por algo que possa recusá-las, registrar o que acontece, revisar. Os produtos e a sintaxe deles mudam com frequência. Quando um produto real é citado, é citado como exemplo, e o comportamento dele é descrito em termos que devem valer entre versões.

Uma nota sobre a certeza

Procurei dizer quais das minhas recomendações são firmes e quais são preferências. Algumas são firmes porque o custo de errar é a exposição de dados de clientes ou a perda do ledger, e digo isso quando é o caso. A maioria são julgamentos, e uma equipe criteriosa poderia pesá-los de outra forma, conforme o tamanho dela, seus reguladores, seus clientes e a frequência com que recebe pages. Onde não souber, direi “não sei” e nomearei as variáveis que resolveriam a questão.

Se você discordar de uma recomendação, gostaria que fosse possível encontrar a premissa com a qual você discorda. É por isso que cada capítulo explicita seu raciocínio e seus custos. Um design que você entende e adaptou às suas próprias restrições vale mais do que um que você copiou de um livro.

O Capítulo 1 começa pela própria porta.

PARTE I

Fundamentos

Capítulo 1 — A porta que ninguém projetou

Considere a Ledgerly em seus primeiros anos. Um cliente em potencial envia um questionário de segurança, e uma das linhas parece rotineira: “Liste todas as pessoas e sistemas que podem se conectar ao seu banco de dados de produção”. Alex, que cuida dos bancos de dados junto com outras atribuições, pega o questionário em uma tarde tranquila.

Alex começa pelo gerenciador de senhas, que guarda uma única entrada de administrador que toda a equipe de backend conhece. Em seguida vêm os cinco serviços, cada um armazenando essa mesma senha, e os laptops, onde os clientes desktop de banco de dados a memorizam. Depois a cópia de analytics, uma restauração noturna de um dump sem mascaramento, e alguns dumps mais antigos em um armazenamento compartilhado que ninguém lembra de ter criado. Cada lugar que Alex verifica aponta para outro.

O questionário tem uma segunda linha: “Como vocês sabem o que essas pessoas e sistemas fizeram?” Alex abre o log do banco de dados e encontra o mesmo nome de papel em todas as linhas, quer a instrução tenha vindo de um serviço, de um engenheiro corrigindo uma linha ou de um relatório que deveria estar lendo de uma réplica. A resposta honesta à primeira pergunta é um amontoado de lugares, e a resposta honesta à segunda é que a Ledgerly não sabe.

Alex não precisa de uma senha melhor. Alex precisa saber de que o acesso é feito e como seria uma resposta curta e defensável. Essa é a pergunta deste capítulo: de que é feito o acesso a um banco de dados em produção, e o que a Ledgerly precisaria para responder a esse questionário sem uma semana de investigação?

Este capítulo descreve o estado inicial da Ledgerly, como faz o Capítulo 2, e ainda não acrescenta nenhum passo do estado-alvo. A função dele é dar o vocabulário e o mapa para os passos que vêm a seguir.

O problema e por que ele é difícil

O acesso a um banco de dados em produção raramente é projetado. Ele se acumula. O primeiro serviço precisa de uma entrada, então alguém cria um papel de administrador e uma senha. O segundo serviço a reutiliza porque funcionou para o primeiro. Um engenheiro precisa olhar uma linha, e a senha vai parar em um cliente desktop. Um job agendado precisa de dados e recebe a mesma connection string que todo o resto. Nenhum desses passos é uma decisão sobre acesso, e a soma deles é a porta que o título do capítulo nomeia.

Veja como é a porta da Ledgerly quando Alex começa a contar. A senha de administrador compartilhada fica em um gerenciador de senhas da equipe e, suponha, a título de ilustração, que cerca de vinte pessoas a conheçam. O papel de administrador que ela destranca é proprietário de todas as tabelas. Os cinco serviços se conectam com esse papel, usando a mesma senha armazenada na configuração de cada serviço, e o pipeline de deploy também executa as migrações com ele. Os engenheiros chegam ao primary a partir dos laptops, pela VPN da empresa, com clientes que memorizam a senha e fazem commit de cada instrução assim que ela roda. O acesso é permanente: ninguém jamais o perde, inclusive quem mudou de equipe há um ano.

As réplicas aceitam o mesmo login de administrador, e as consultas de relatório às vezes rodam no primary. A cópia de analytics é uma restauração de um dump sem mascaramento, e os arquivos de dump continuam por lá. Um fornecedor externo de contabilidade nunca se conecta a nenhum banco de dados da Ledgerly, mas recebe exports de dados de faturas e do ledger, e a etapa de export do reporting-job grava esses arquivos com o login de administrador compartilhado. O log do banco de dados mostra um único papel. Nada nesse arranjo registra qual pessoa executou qual instrução.

É tentador tratar isso como um problema de disciplina: dizer às pessoas que tenham cuidado, trocar a senha com mais frequência. Não começaria por aí. Uma regra que só funciona quando todos se lembram dela falha na primeira vez que a pessoa on-call estiver cansada. O sistema dá a todo principal as mesmas capacidades e o mesmo nome, de modo que quem

quer ter cuidado não tem com o que ter cuidado: nenhum papel mais restrito para escolher, nenhuma forma de dizer “somente leitura” ao banco de dados, nenhuma sessão que carregue um nome.

O mesmo vale para o remédio a que as pessoas recorrem primeiro, que é tratar melhor a senha: guardá-la em um vault, rotacioná-la, entregá-la apenas a quem pedir. Isso é distribuição de credenciais, e vale a pena fazê-lo, mas responde a uma só pergunta: quem detém o secret. O acesso a bancos de dados em produção não é um problema de distribuição de credenciais; é um problema de autorização, execução e responsabilização. A autorização pergunta o que cada principal pode fazer e por qual rota. A execução pergunta como cada instrução chega aos dados, se algo a inspeciona antes e até onde ela pode alcançar. A responsabilização pergunta se a evidência vincula cada instrução a uma pessoa e a uma requisição. Uma senha perfeitamente armazenada resolve quem detém o secret e nada mais: quem a detém ainda pode executar uma instrução que ninguém inspecionou, e o log ainda pode mostrar um nome compartilhado.

É por isso que esta é uma questão de engenharia, e não de política. Uma política enuncia o que deveria acontecer. Um design determina o que pode acontecer e o que deixa registro quando acontece. O restante deste capítulo toma a forma de um problema de design: do que o acesso é composto, o que queremos que ele faça e onde cada parte é tratada neste livro.

O acesso tem cinco partes

Antes de julgar qualquer design, ajuda ter uma definição precisa do que está sendo projetado. Considero útil dizer que o acesso a um banco de dados em produção tem cinco partes: uma identidade, um grant, um caminho, uma execução e um registro. A tabela diz a que pergunta cada parte responde e a qual pilar da tese do livro ela pertence.

Parte	Pergunta	Pilar
Identidade	Quem ou o que está agindo: uma pessoa ou uma instância de workload?	Responsabilização
Grant	O que esse principal pode fazer, afinal: papéis, privilégios, teto?	Autorização
Caminho	Como a conexão chega: alcance de rede, gateway, pooler?	Autorização (onde)

Parte	Pergunta	Pilar
Execução	Como uma instrução chega aos dados? Quem ou o que a envia, se ela é inspecionada antes de rodar, até onde pode alcançar (transação, row guard, timeout) e como é confirmada com commit.	Execução
Registro	Que evidência permanece, e ela vincula a instrução à pessoa, à requisição e ao resultado?	Responsabilização

Os três pilares se distribuem pelas partes assim. A autorização é grant mais caminho, decidida por requisição. A execução é o que acontece com cada instrução. A responsabilização é identidade mais registro. Uma senha toca apenas a primeira delas, e somente na porta.

A **identidade** é a afirmação verificada de que um principal é quem diz ser. Um **principal** é qualquer coisa que se autentica: uma pessoa, um serviço, um job agendado, uma integração de fornecedor. Principal é a palavra a usar quando uma afirmação vale tanto para pessoas quanto para workloads, e este livro a usa assim do começo ao fim. Um papel não é uma identidade. Um **papel** é um conjunto nomeado de privilégios dentro do banco de dados e pode ser compartilhado por muitos principais, que é exatamente o que o papel de administrador compartilhado faz.

O **grant** é o que o principal pode fazer depois que entra. Aqui, um grant é o ato, e o registro resultante, de dar um privilégio a um papel, ou um papel a um **login**; o Capítulo 3 amplia a definição. Um **privilégio** é uma coisa específica, como ler uma tabela, e o login é a conta no nível do banco de dados com a qual um principal se conecta. O **caminho** é a rota que uma conexão percorre para chegar ao banco de dados: uma rede, um proxy, um gateway, um laptop.

A **execução** é como uma instrução chega aos dados: quem ou o que a envia, se algo a inspeciona antes de rodar, o que limita seu efeito e como ela é confirmada com commit. Um grant diz que um papel pode fazer update em `invoices`. Ele não diz que esse update deveria atingir uma só linha, que alguém olhou antes, nem que ele roda dentro de uma transação que pode sofrer rollback. No estado inicial, nada responde a essas perguntas: o cliente desktop envia o que for digitado e faz commit na hora. O Capítulo 9 trata dessa parte.

O **registro** é a evidência posterior, a resposta a “quem fez o quê e quando?” e, no caso de uma escrita, “em quais linhas, sob qual requisição?”

Cada parte pode falhar por conta própria, mas cada uma depende das outras para ser útil. Sem uma identidade distinta, um grant preciso não tem a quem se prender. Sem um caminho escolhido, uma identidade pode ser contornada por uma rota que nunca pede uma. Sem uma execução controlada, um grant restrito ainda permite uma instrução mais ampla do que alguém pretendia. Sem um registro, não há como saber se as outras quatro funcionaram. Essa dependência é a razão pela qual corrigir apenas uma parte, geralmente a senha, deixa o restante do problema de pé.

Um teste: o log consegue responder à pergunta?

Uma forma de descobrir quais partes estão faltando é fazer uma pergunta ao log e observar onde a resposta para. O pseudocódigo a seguir rastreia uma única instrução pelas cinco partes, usando apenas o que o estado inicial registra.

```
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 }
```

No estado inicial da Ledgerly, `principalsUsing("admin")` devolve cerca de vinte pessoas, cinco serviços e o pipeline de deploy, de modo que a função para na primeira verificação para toda instrução. Isso não é um bug do log. O log é fiel ao que viu, que é um único login. A lacuna está no design que colocou muitos principais atrás de um único login. Corrija a identidade, e a mesma função pararia em seguida na verificação de execução. O incidente inaugural é o caso para o qual ela existe: uma correção de uma linha cuja cláusula `WHERE` atinge um lote inteiro tem `rowsAffected` muito acima de `expectedRows`, nunca foi revisada e nunca mostrou sua contagem antes do `COMMIT`, de modo que para em “execution” mesmo que a pessoa seja conhecida e o grant o permitisse. Uma instrução que passa por essa verificação ainda pode parar em “record” se nada a vincular a uma requisição e a um resultado.

Aplicações e pessoas, uma só pergunta

Quando ouvem “acesso a banco de dados”, as pessoas costumam imaginar engenheiros com ferramentas de consulta. Essa imagem é incompleta de um jeito que importa. Boa parte do tráfego para o primary da Ledgerly vem de software: o `billing-api` atendendo o aplicativo web, o `invoice-worker` enviando lembretes, o `payments-service` falando com provedores de pagamento, o `reporting-job` rodando à noite e o `admin-console` agindo em nome da equipe de suporte. O software se conecta com mais frequência, por mais tempo e com menos supervisão do que qualquer pessoa.

Serviços e pessoas falham de maneiras diferentes, e por isso a Parte II trata das aplicações e a Parte III trata das pessoas. O contraste é concreto:

	Serviços	Pessoas
Frequência	Constante, 24 horas por dia	Ocasional, em rajadas
Supervisão	Nenhuma enquanto roda	Alguém acompanha o resultado
Duração da credencial	De curta duração e emitida por um broker (Capítulo 4)	Uma sessão ou uma janela aprovada (capítulos 7 e 8)
Falha típica	Um secret vazado, um connection pool esgotado, um deploy que aponta para o banco de dados errado	Uma instrução mais ampla do que o pretendido, executada à mão sob pressão

Ambos se apoiam nos fundamentos da Parte I.

Há mais um tipo de principal que é fácil de deixar passar: um serviço agindo em nome de uma pessoa. O `admin-console` é usado pela equipe de suporte, mas, no estado inicial, conecta-se com o login de administrador compartilhado como todo o resto, de modo que o banco de dados vê o console, nunca a pessoa que clicou. Sam, do suporte, olha um cliente por vez, e o design deixa o console alcançar os dados de todos os clientes sob um nome que não é de ninguém. O Capítulo 3 mostra como o console repassa a identidade da pessoa junto com cada ação, e o Capítulo 14, como os registros de auditoria são unidos, para que a pessoa sobreviva ao salto.

Segurança e confiabilidade são a mesma pergunta

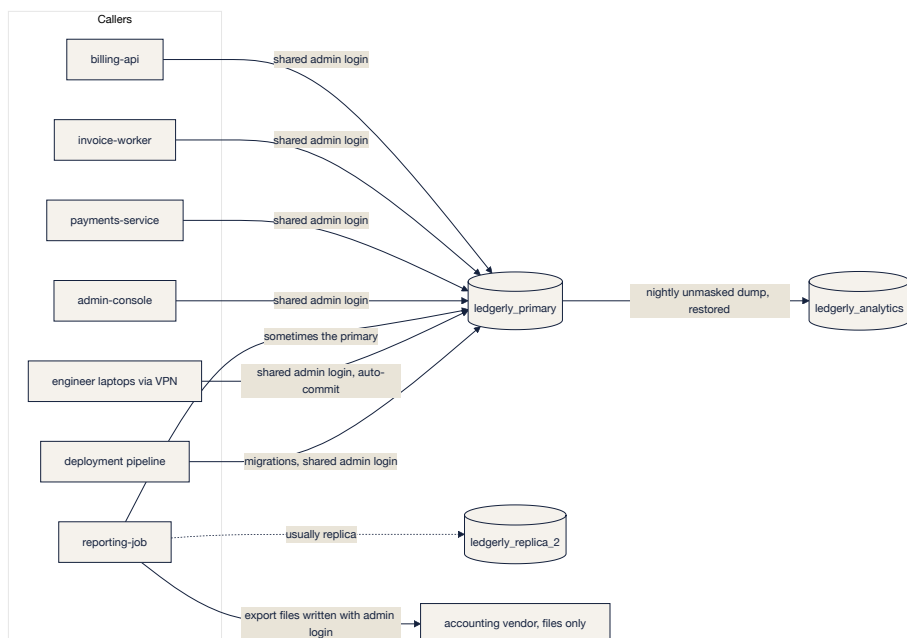
As equipes costumam manter duas listas. Uma trata de segurança: quem poderia roubar dados, quem poderia abusar do acesso. A outra trata de operações: o que poderia derrubar o banco de dados ou corrompê-lo. Em geral, têm responsáveis diferentes e são revisadas em momentos diferentes. Para o acesso, as uniria, porque as perguntas são as mesmas.

Tome três eventos que parecem não ter relação. Uma instrução com uma cláusula `WHERE` errada ou ausente atualiza todas as faturas em vez de uma. Um cliente em um laptop deixa uma conexão aberta e ocupa uma vaga a noite toda. Um dump do banco de dados de produção é copiado para um armazenamento compartilhado e lido por alguém que não tinha nada que lê-lo. O primeiro é um problema de integridade dos dados, o segundo, de disponibilidade, o terceiro, de confidencialidade. Ainda assim, cada um faz ao design a mesma pergunta: por que um único principal conseguiu fazer algo tão amplo, sem uma rota mais restrita, um limite ou um registro?

Os três eventos compartilham uma causa de design: nenhum estreitamento de grant, de duração ou de caminho, e nada que mostrasse o ocorrido depois. A cláusula `WHERE` errada é, em grande parte, uma lacuna de grant e de execução; a conexão aberta, uma lacuna de duração e de limite; e o dump copiado, uma lacuna de caminho. Restringir o grant, limitar a duração, escolher o caminho, limitar a instrução e manter o registro ajudam nos três eventos ao mesmo tempo. É por isso que um único design atende às duas listas. Um papel que não pode ler `payment_tokens` protege os cli-

entes de um vazamento, e um papel que não pode escrever em `ledger_entries` protege o ledger de uma atualização em lote errada. Ambos nascem do mesmo hábito de conceder de forma restrita.

O diagrama abaixo mostra o estado inicial da Ledgerly. Observe quantas setas terminam no mesmo nó e a única seta que sai dele em direção à cópia de analytics.



Duas coisas se destacam. Primeiro, todo caminho até o primary carrega o mesmo nome, de modo que o banco de dados não distingue um worker de uma pessoa. Segundo, os dados também saem por uma rota lateral: a restauração noturna copia tudo, inclusive colunas de que nenhum analista precisa, para um armazenamento com público próprio.

A gama de formas de entrar

Entre a senha compartilhada e um design totalmente controlado há uma gama de padrões, e os capítulos seguintes situam cada um nela. O diagrama vai do menor ao maior controle sobre cada instrução. Observe o que cada passo acrescenta e onde aparece a primeira verificação no nível da instrução.



Cada passo para a direita acrescenta uma propriedade e custa algo. Um bastion estreita o caminho, mas a sessão do outro lado continua sem supervisão. Um session proxy vincula uma conexão a uma pessoa e a registra, mas entrega uma sessão em que se pode digitar qualquer coisa que o grant permita. Um gateway que medeia a execução olha cada instrução antes de ela rodar e pode recusá-la. Um caminho de produto elimina a sessão por completo: a pessoa pede uma operação definida, como “reenviar esta fatura”, e nunca escreve SQL. Nada aqui diz que o padrão mais à direita é sempre o melhor; ele também é o que mais exige construção e o mais lento para mudar. O Capítulo 7 compara os quatro primeiros em caminhos concretos, o Capítulo 9 trata das verificações no nível da instrução, e o capítulo final pontua os padrões por pilar.

Como é um bom acesso

Escreveria o objetivo antes de escolher qualquer ferramenta. Uso cinco propriedades, e cada uma tem um preço.

O **privilegio mínimo** é a prática de conceder apenas o que uma tarefa precisa, pelo tempo de que ela precisa. O preço é o esforço de descobrir o que cada tarefa precisa e o atrito quando uma tarefa acaba precisando de mais. Uma equipe que o adota sem uma forma rápida de pedir mais verá as pessoas darem a volta nele.

A **vida curta** significa que o acesso termina sozinho. Um grant que dura uma hora não exige que alguém se lembre dele. Compare com o **acesso permanente**, que existe continuamente, seja usado ou não. Um grant permanente é uma promessa de que ninguém precisará pedir de novo, e é conveniente no primeiro mês e caro em todos os meses seguintes, porque ninguém lembra o que a promessa cobria. O custo da vida curta é pedir mais vezes e uma expiração que pode chegar no meio de um reparo.

A **observabilidade** significa que todo uso do acesso deixa um registro que nomeia o principal, e que o registro fica em um lugar que o principal não pode alterar. O custo é armazenamento e, mais importante, atenção: um

log que ninguém lê é teatro de segurança, pois custa armazenamento e não dá nenhuma proteção. O Capítulo 14 volta a isso.

A **reversibilidade** tem dois sentidos, e ambos importam. O acesso deve ser fácil de retirar, e um erro cometido com o acesso deve poder ser desfeito ou, ao menos, contido. A revogação precisa ser rotineira, e desfazer um erro exige correções revisadas e backups que restauram. Uma forma barata de reversibilidade é executar uma escrita dentro de uma transação explícita e mostrar a contagem de linhas antes do `COMMIT`: um update que deveria atingir uma linha e informa, digamos, quatrocentas é interrompido enquanto ainda pode sofrer rollback. O Capítulo 9 mostra como um gateway torna isso o padrão. O custo é o tempo gasto em ensaios.

O **baixo atrito** é a propriedade que as equipes esquecem de escrever. Se o caminho legítimo é lento, as pessoas encontrarão o ilegítimo, e com razão, porque o incidente das 2 da manhã não espera. O custo do baixo atrito é que ele restringe as outras quatro: todo controle precisa ser barato o bastante para ser usado na pior hora da semana.

Essas cinco puxam em direções opostas. A vida curta acrescenta atrito. A observabilidade acrescenta um salto: uma sessão que é gravada precisa passar por algo que a grave, e esse algo pode ser lento ou estar fora do ar. O privilégio mínimo multiplica os papéis. Não acredito que exista uma configuração que maximize as cinco. A pergunta útil não é “quem deveria ter acesso?”, mas “quais instruções podem rodar, por meio de quê e com qual registro?” Cada capítulo seguinte responde a uma fatia dela e diz quanto a resposta custa.

Um primeiro inventário

Antes de adotar qualquer controle, anotaria o que existe. O esboço a seguir mostra a forma de um inventário para o estado inicial da Ledgerly. A ideia é ter uma linha por porta, com um responsável nomeado e um motivo, de modo que uma linha sem responsável fique visível como lacuna.

```
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
```

Preencha um bloco por porta: cada serviço, o job agendado, o pipeline de deploy, os laptops, a restauração de analytics, os dumps no armazenamento compartilhado e os arquivos de export. A maioria das equipes descobre que as linhas `owner` e `ends` são as que ficam em branco. Um inventário como este não é um plano de projeto. É o que transforma “quem consegue chegar ao primary?” de um dia de investigação em um documento que se pode ler.

Um mapa deste livro

Os capítulos seguintes tratam, cada um por vez, de cada parte do acesso e de cada falha descrita acima. A tabela liga cada pergunta levantada neste capítulo ao lugar onde o livro a responde.

A pergunta	Pilar	Onde é respondida
De quem e do que estamos protegendo os dados, e contra que tipos de eventos?	Os três	Capítulo 2
Como damos a cada principal uma identidade distinta e o papel mais restrito que funciona?	Autorização	Capítulo 3
Como os serviços obtêm credenciais sem uma senha compartilhada em um arquivo?	Responsabilização (identidade)	Capítulo 4
Como os limites de conexão e os timeouts se tornam uma política que protege o banco de dados?	Autorização e execução (limites)	Capítulo 5
Como as leituras chegam às réplicas, e o que acontece com as outras cópias dos dados?	Autorização	Capítulo 6
Por quais caminhos as pessoas chegam ao banco de dados, e o que fica em cada caminho?	Autorização (caminho), responsabilização	Capítulo 7

A pergunta	Pilar	Onde é respondida
Como alguém obtém acesso de escrita para uma tarefa, e como ele termina?	Autorização	Capítulo 8
Como uma instrução é verificada antes de rodar, e como é confirmada com commit?	Execução	Capítulo 9
Como as consultas ad hoc e as correções de dados continuam revisáveis?	Execução	Capítulo 10
Quem altera o esquema, e com qual acesso?	Execução	Capítulo 11
Quais dados precisam de proteção extra, e como são mascarados ou retidos?	Autorização	Capítulo 12
E os backups, os exports e as outras rotas que contornam os controles?	Autorização (caminho)	Capítulo 13
Como mantemos um registro que se sustenta, e como o revisamos?	Responsabilização	Capítulo 14
O que muda durante um incidente?	Os três, sob pressão	Capítulo 15
O que fazemos primeiro, e como mantemos isso em dia?	Os três	Capítulo 16

O livro tem cinco partes: fundamentos, acesso das aplicações, acesso humano, proteção dos dados e de suas cópias, e a comprovação e a operação de tudo isso. Cada capítulo nomeia, na abertura, o pilar a que serve. Você não precisa ler em ordem. Se o seu problema imediato é a senha compartilhada em cinco arquivos de configuração, o Capítulo 4 é o lugar para começar, mas o Capítulo 3 vai poupar você de criar o próximo papel compartilhado por acidente.

Trade-offs e custos

O primeiro custo é **tempo e atenção de um grupo pequeno**. Alguém precisa montar o inventário, definir papéis, configurar credenciais e conduzir revisões. Em uma equipe pequena, isso muitas vezes é o trabalho em meio período de uma pessoa, e compete com o trabalho de produto. Se o trabalho nunca entra na agenda, o design fica no papel. Prefiro ver um primeiro passo pequeno e concluído a um plano grande.

O segundo custo é o **atrito para quem está on-call**. Todo controle coloca um passo entre uma pessoa cansada e um reparo. Papéis restritos significam que a primeira tentativa às vezes falha com um erro de privilégio na pior hora. O acesso de curta duração significa pedir de novo. Decida com antecedência qual é o caminho rápido e ensaie-o. Um controle que é contornado em todo incidente não é um controle; é um atraso antes do contorno.

O terceiro custo são os **novos sistemas que precisam estar disponíveis quando necessários**. Um gateway, um armazenamento de secrets e um log store são, cada um, mais uma coisa que pode estar fora do ar. Antes de adotar um, pergunte o que acontece com quem precisa do banco de dados durante essa indisponibilidade, e tenha uma resposta pronta.

Há também casos em que o design completo é o lugar errado para começar. Um protótipo de um único serviço, sem dados de clientes, operado por duas pessoas, não precisa de um gateway nem de um calendário de revisões. Ele precisa de um login próprio, separado do de qualquer pessoa, porque isso é barato, e de uma nota dizendo o que levaria a equipe a revisar a questão: o primeiro cliente, o terceiro engenheiro ou a primeira integração externa. Não sei qual é o momento certo para a sua empresa. Depende, e eis do quê: de quão sensíveis são os dados, de quantos principais os tocam e de com que rapidez uma instrução ruim pode causar um dano que você não consegue reverter.

Por fim, as propriedades do bom acesso têm um limite. Elas reduzem o raio de impacto e tornam o comportamento visível. Não tornam um banco de dados seguro, e nenhum capítulo deste livro afirma o contrário. O **raio de impacto** é quanto pode ser afetado se um principal, uma credencial ou uma ação der errado, e o trabalho deste livro é tornar esse raio menor e mais visível, não torná-lo zero.

Erros comuns

Estas são armadilhas de design, não falhas pessoais. Cada uma é tentadora por um motivo, e cada uma tem um remédio que não depende de ninguém ser mais cuidadoso.

- **Tratar a troca de senha como o remédio.** Uma senha nova dá a sensação de progresso e elimina o conhecimento de um ex-colega. Mas,

se, digamos, vinte pessoas e cinco serviços ainda a compartilham, o design não mudou. A correção é um design em que ninguém precisa compartilhar uma senha, que é o que os capítulos 3 e 4 constroem.

- **Corrigir o grant e esquecer o caminho.** Papéis restritos só ajudam se as conexões forem obrigadas a usá-los. Se os laptops ainda alcançam o primary com um antigo login de administrador, os papéis restritos são opcionais. Faça o inventário de todos os caminhos, não apenas de todos os papéis.
- **Deixar as aplicações de fora.** As revisões de “quem tem acesso ao banco de dados” muitas vezes listam apenas pessoas, porque as pessoas aparecem em um organograma. Os serviços detêm as credenciais mais poderosas e as usam mais. Inclua todo workload, todo job agendado e todo script.
- **Fazer o caminho seguro mais lento que o inseguro.** Se ler a produção exige um ticket e um dia, enquanto a senha compartilhada exige, digamos, dez segundos, a senha compartilhada vence todas as vezes. Busque um caminho seguro que não seja mais lento.

Aviso: Um dump sem mascaramento em um armazenamento compartilhado é a rota mais direta do estado inicial até os dados dos clientes, e está aberto a qualquer pessoa que consiga ler a pasta. Trate a localização e a remoção desses arquivos como parte do primeiro inventário, antes de qualquer redesenho de papéis.

Pontos-chave

- O acesso a bancos de dados em produção não é um problema de distribuição de credenciais; é um problema de autorização, execução e responsabilização, e uma senha responde apenas à primeira pergunta, na porta. O acesso é um design de cinco partes, a saber Identidade, Grant, Caminho, Execução e Registro, e uma fraqueza em qualquer uma delas compromete as outras.
- Uma conta de administrador compartilhada não é uma decisão que alguém tomou; ela se acumula, e o remédio é um design em que ninguém precisa compartilhar uma senha.

- As aplicações se conectam com mais frequência e com menos supervisão do que as pessoas, de modo que qualquer tratamento do acesso precisa cobrir serviços, jobs e ferramentas como principais, ao lado dos humanos.
- Segurança e confiabilidade operacional são uma só pergunta quando se trata de acesso, porque uma cláusula `WHERE` errada, uma conexão vazada e um dump copiado têm uma causa de design em comum: nenhum estreitamento de grant, de duração ou de caminho, e nenhum registro.
- O bom acesso tem cinco propriedades, que são privilégio mínimo, vida curta, observabilidade, reversibilidade e baixo atrito, e elas puxam em direções opostas, de modo que todo controle deve ser apresentado junto com o que custa.
- O primeiro passo é um inventário com uma linha por porta, incluindo os serviços, os laptops e as cópias dos dados, em que um responsável ausente ou uma data de término ausente aparece como uma lacuna visível.
- Os controles reduzem o raio de impacto e tornam o comportamento visível, mas não tornam um banco de dados seguro, e o objetivo honesto é um raio menor e uma visão mais clara.

O próximo capítulo passa da forma do acesso para as razões de protegê-lo. O Capítulo 2 pergunta para quem e para quê servem os controles, separa em três tipos os eventos que podem dar errado e escreve um modelo de ameaças para a Ledgerly ao qual o restante do livro possa voltar.

Capítulo 2 — Um modelo de ameaças para os dados de produção

Imagine que o contato de segurança da Ledgerly envia à equipe de plataforma uma pergunta curta antes de uma revisão trimestral: “De quem estamos protegendo o banco de dados de produção?” Alex, que cuida dos bancos de dados, responde em um minuto: de gente de fora, principalmente. Então Alex faz uma pausa, porque a resposta não combina com os próprios tickets da equipe no último ano. Uma requisição de suporte que mexeu no cliente errado. Uma consulta de relatório que deixou a escrita de faturas lenta durante uma tarde. Um dump restaurado em um laptop por um dia de depuração, e ninguém foi lembrado de apagá-lo. Um export para um fornecedor que saiu com uma coluna a mais do que o pretendido.

Nenhum desses casos envolveu alguém de fora. Nenhum envolveu má intenção. Todos envolveram uma porta aberta a alguém que tinha uma tarefa razoável, e um papel que permitia mais do que a tarefa exigia. A primeira resposta de Alex foi a que soa como segurança. A segunda lista é a que descreve o sistema.

Este capítulo descreve a Ledgerly no estado inicial, como fez o Capítulo 1, e não acrescenta nenhum controle próprio. O que ele acrescenta é um modelo de ameaças escrito, a referência contra a qual se escolhe cada controle posterior do livro. Ele organiza as ameaças em torno dos três pilares: o que um principal pode fazer, como cada instrução chega aos dados e que evidência permanece. A pergunta que ele responde é a que Alex não conseguiu responder em um minuto: quem e o que pode prejudicar os dados, por quais portas, e até onde o dano se propaga?

O problema e por que ele é difícil

O desenho de acesso sem um modelo de ameaças tende a seguir a preocupação mais barulhenta. Se for um atacante externo, a equipe constrói um perímetro: uma VPN, uma regra de firewall, uma allowlist de rede. Isso tem importância. Mas um perímetro responde a “quem pode chegar perto do banco de dados?” e não diz nada sobre o que uma pessoa ou um serviço pode fazer depois de entrar. No estado inicial da Ledgerly, qualquer pessoa na VPN com a senha compartilhada pode fazer tudo o que o papel de administrador pode fazer, ou seja, tudo.

A alternativa óbvia é uma lista de controles tirada de um checklist e aplicada em ordem. Um checklist não sabe quais das suas portas carregam mais risco. Ele vai pedir a rotação de uma senha que ninguém conseguiria rotacionar sem uma indisponibilidade, e não dirá nada sobre a restauração noturna que copia toda coluna Restricted para um armazenamento com um público muito maior. Controles escolhidos sem um modelo são escolhidos por disponibilidade e por moda. Alguns reduzem a exposição. Outros são teatro de segurança, atrito sem uma redução correspondente do risco, e a equipe não sabe distinguir uns dos outros porque nunca escreveu para que servia o controle.

Um modelo escrito é necessário por três razões. Primeiro, as portas são objetos técnicos: um login, um caminho de rede, um arquivo em armazenamento compartilhado, uma variável de pipeline. Uma política pode dizer “o acesso deve ser apropriado”, mas só um desenho decide o que um login alcança. Segundo, os maiores riscos vêm da interação de escolhas comuns, como descreve o Prefácio: uma senha compartilhada, um cliente com auto-commit e um papel que é proprietário de todas as tabelas. Nenhuma escolha isolada parece errada, então uma revisão de escolhas isoladas não encontrará o problema. Terceiro, o modelo precisa ser barato o bastante para manter, ou será escrito uma vez e deixado à deriva.

Um modelo de ameaças, no sentido usado aqui, é pequeno. Ele lista quem e o que pode agir sobre os dados, por qual caminho, o que pode dar errado, até onde o efeito se propaga e qual controle trata do assunto. Não é uma pontuação de risco nem uma previsão. É um modo de fazer a mesma pergunta sobre cada porta, para que as respostas possam ser comparadas.

Nota: O livro descreve as ameaças apenas na medida em que elas justificam um controle. Este capítulo nomeia o que pode dar errado, como o vazamento de uma credencial ou uma consulta que cai no banco de dados errado, e não descreve como alguém faria isso acontecer. Se mais detalhes não mudariam o desenho, eles ficam de fora.

Quem e o que chega ao banco de dados

O Capítulo 1 disse que o acesso a um banco de dados de produção tem cinco partes: uma Identidade, um Grant, um Caminho, uma Execução e um Registro. Um modelo de ameaças começa listando os principais e faz as cinco perguntas a cada um. A Ledgerly tem nove tipos de atores que vale listar. Dois deles, o fornecedor e o atacante externo, não são principais no sentido estrito, e é por isso que pertencem à lista: é fácil deixá-los de fora.

Serviços

Os serviços (`billing-api`, `invoice-worker`, `payments-service` e `admin-console`, com o `reporting-job` tratado em seguida como um job) são os maiores usuários do banco de dados. Eles rodam sem supervisão, guardam suas credenciais na configuração e se comportam de forma tão correta quanto o seu código. Um serviço pode se comportar mal sem que ninguém faça nada errado: um loop de retry que abre conexões até o limite ser atingido, uma consulta que varre uma tabela porque um índice foi removido, um deploy que aponta um build de staging para o primary.

No estado inicial, todo serviço se conecta com o mesmo login de administrador, de modo que o raio de impacto de cada serviço é o banco de dados inteiro, embora `billing-api` e `invoice-worker` precisem, cada um, de apenas um punhado de tabelas. Um defeito no menor serviço tem o alcance do maior.

Jobs em lote

O `reporting-job` merece sua própria linha porque um job roda de acordo com um agendamento, muitas vezes à noite, sem ninguém observando. Ele tende a ler muitos dados, manter transações longas e produzir arquivos. No estado inicial, às vezes ele lê o primary, e a etapa de export grava

os arquivos do fornecedor com o login de administrador compartilhado. Seu comportamento normal já se parece com acesso em massa, o que torna o comportamento anormal mais difícil de perceber do que o de uma pessoa.

Engenheiros de on-call

O rodízio de on-call é o que sofre mais pressão para ter acesso amplo, e com bom motivo: um page durante a noite não é o momento de esperar uma aprovação que ninguém está acordado para dar. No estado inicial, essas pessoas usam a senha de administrador compartilhada por meio de clientes de desktop que a memorizam e fazem commit a cada instrução. A combinação de urgência para consertar e nenhum atrito para alterar qualquer coisa faz deste o grupo em que os acidentes são mais prováveis, e o grupo que o desenho consegue proteger melhor.

Suporte

A equipe de suporte usa o `admin-console` para consultar um cliente por vez. A ameaça aqui é o escopo, não a malícia. Uma ferramenta que consegue exibir todos os clientes, apoiada em um papel que consegue ler todas as colunas, responde a uma pergunta sobre uma única fatura com um sistema que poderia mostrar todas as faturas. A pergunta de design é como tornar a capacidade ampla indisponível quando só a estreita é necessária.

A equipe de dados

A equipe de dados trabalha em `ledgerly_analytics`, que no estado inicial é uma restauração noturna de um dump sem mascaramento: todas as colunas Restricted e Confidential, acessadas por um grupo cujo trabalho é olhar dados em massa. O acesso dessa equipe é legítimo. A preocupação é que a cópia tem um público mais amplo, um manuseio mais frouxo e menos logs do que o primary, de modo que um controle no primary não protege o que está na cópia.

Fornecedores e integrações

O fornecedor externo de contabilidade nunca se conecta a um banco de dados da Ledgerly. Ele recebe arquivos. Isso muda a forma da ameaça: a exposição não é um login, e sim um export, e seu raio de impacto são as

colunas que o export contém e por quanto tempo a cópia existe do lado do fornecedor. As perguntas relevantes são o que o arquivo contém, quem é o responsável por ele depois e se alguém perceberia que o arquivo ganhou uma coluna.

CI/CD

O pipeline de deploy também é um principal, ainda que seja infraestrutura e não uma pessoa ou um serviço. No estado inicial, ele executa migrações com o mesmo login de administrador, porque o papel de administrador é proprietário de todas as tabelas e nada mais pode alterá-las, e a credencial fica na configuração do pipeline. Um pipeline que pode alterar qualquer tabela e guarda uma credencial permanente é, ao mesmo tempo, um risco de comprometimento e um risco de erro, já que uma migração ruim roda com todos os privilégios do proprietário.

Equipe de plataforma e DBAs

O pequeno grupo de plataforma opera os bancos de dados, e é o único ator que pode alterar os próprios controles: papéis, grants, a entrada pela qual os humanos passam, o log. No estado inicial, ele usa o login de administrador compartilhado como todo mundo, de modo que nada separa o seu trabalho do de qualquer outro. No desenho-alvo, ele detém um papel para gerenciar o acesso, `ledgerly_access_admin`, que três principais usam por meio de seus próprios logins. Isso não elimina a ameaça; apenas a desloca. Um grant equivocado ou uma configuração de log desativada por quem tem esse papel muda o que significa cada outra linha do modelo.

Por essa razão, o modelo lista o **plano de controle** como um ativo por direito próprio: o access gateway, o credential broker, o `ledgerly_access_admin` e o armazenamento de auditoria. Os dados são protegidos pelos controles, e os controles são protegidos por um conjunto muito menor de pessoas e sistemas. A exposição é pequena em volume e grande em efeito, e as perguntas são as mesmas de qualquer porta: quem consegue chegar a ela, o que ela poderia alterar e quem perceberia.

O atacante externo

Um atacante externo não é um principal até obter uma credencial, então a maneira útil de modelá-lo é pelo que ele precisaria obter, não pelo que poderia fazer. As perguntas de quem defende são: quais portas aceitam uma credencial que poderia vazar, o que ela alcançaria e como descobriríamos? As respostas vêm dos outros oito atores. Uma credencial vazada herda o raio de impacto de quem a possuía.

Três classes de ameaça

Os mesmos atores podem causar dano de três maneiras diferentes, e os controles diferem para cada uma. Misturar as classes leva a controles que não se encaixam: uma aprovação de intenção, desenhada para o uso indevido, faz pouco contra um acidente, porque quem aprova concorda que a tarefa é legítima e a instrução continua dando errado.

Erros

Um erro é uma ação que pretendia ser correta e não foi: uma instrução executada no ambiente errado, uma condição que corresponde a mais linhas do que o pretendido, uma consulta pesada demais para o primary, um script apontado para a produção por uma configuração desatualizada. O ator é um principal fazendo o seu trabalho.

Esperaria que essa classe fosse a mais frequente na maioria das equipes, por uma razão estrutural e não estatística. O uso indevido exige uma pessoa com um motivo, e o comprometimento exige um atacante com uma oportunidade, mas um erro exige apenas uma pessoa com uma tarefa. Não tenho um número para dizer quanto os erros são mais frequentes, e não confiaria em um se me oferecessem. A conclusão para o desenho vale de qualquer forma: os controles voltados a erros protegem contra a classe que toda equipe tem, quer ela enfrente ou não as outras duas.

A melhor maneira de tratar os erros é reduzir o que uma única instrução consegue alcançar, já que não se pode pedir que as pessoas sejam cuidadosas para sempre. Papéis que não conseguem tocar tabelas de que não precisam (Capítulo 3), timeouts que interrompem uma consulta descontrolada (Capítulo 5), análise de cada instrução antes de ela rodar (Capítulo 9),

uma rota distinta para escritas com revisão four-eyes (capítulos 8 e 10) e uma forma revisada de reparar dados depois (Capítulo 10) pertencem a essa classe.

A etapa four-eyes ajuda aqui apenas sob uma condição: quem revisa vê a instrução e o seu efeito esperado. Aprovar a intenção (“corrigir a fatura 1042”) é um controle para a classe do uso indevido, e deixa a classe dos erros intocada. Aprovar a instrução e o número de linhas que ela deveria alterar trata dos erros.

Uso indevido e curiosidade

Uso indevido, aqui, significa um principal com acesso legítimo que o usa para algo fora da tarefa: ler o registro de um cliente por interesse, fazer export de mais do que o trabalho precisa, olhar os dados de colegas. Inclui a simples curiosidade, que provavelmente é a forma mais comum e a mais difícil de classificar como má conduta. Inclui também o raro caso deliberado.

A resposta do desenho não é desconfiar da equipe. Uma equipe não precisa supor má-fé para perceber que um grant permanente torna possível navegar pelos dados e não deixa rastro. Os controles que importam mantêm o acesso ligado a uma tarefa e o tornam visível: papéis somente leitura que retêm as colunas Restricted, views mascaradas dos dados Confidential (Capítulo 12), grants com prazo (Capítulo 8) e um log de auditoria que alguém de fato lê (Capítulo 14). Sem evidência, não se pode responder à pergunta “esse acesso foi apropriado?” para ninguém, inclusive para quem não fez nada de errado.

Comprometimento

Comprometimento significa que uma credencial ou uma máquina está nas mãos de alguém que não é o principal pretendido: uma senha que saiu por um arquivo de configuração, um laptop que foi perdido, uma variável de pipeline lida por algo em que não se deveria confiar. Quem defende não precisa saber como aconteceu para projetar levando em conta que pode acontecer. O desenho controla o que a credencial poderia fazer, por quanto tempo e se alguém ficaria sabendo.

O comprometimento é onde as propriedades de uma credencial mais importam. Uma credencial compartilhada, estática e irrestrita transforma a perda de um laptop na perda de todas as tabelas. Uma credencial restrita a um papel, de curta duração e emitida para uma sessão transforma o mesmo evento em algo limitado e visível. Os capítulos 4 e 7 tratam de como as credenciais são emitidas; o Capítulo 3 trata de quão pouco elas devem carregar; o Capítulo 14 trata de como perceber.

Nota: Classificar um evento muitas vezes é uma questão de julgamento. Uma correção de uma linha que alcança um lote inteiro é um erro. A mesma instrução executada de propósito por alguém sem tarefa é uso indevido, e executada com uma credencial tirada de um laptop é comprometimento. O banco de dados vê a mesma instrução em cada caso, e esse é o argumento a favor de logs que nomeiem um login e de controles que não dependam de adivinhar a intenção.

O que está sendo protegido

Três propriedades dos dados merecem proteção, e o modelo deve dizer qual delas cada ameaça atinge. São as conhecidas confidencialidade, integridade e disponibilidade, e cada uma tem um significado específico na Ledgerly.

A **confidencialidade** diz respeito a quem pode ler o quê. Para a Ledgerly, o peso é desigual. `iban`, `account_holder_name` e `token_value` são Restricted, de modo que sua exposição seria difícil de desfazer. `contact_name`, `contact_email`, `contact_phone` e `billing_address` são dados PII de clientes classificados como Confidential (informações de identificação pessoal). Os valores de `invoices` e `payments` são Internal. Uma ameaça que toca dados Internal merece menos esforço de design do que uma que toca dados Restricted, e o modelo deve mostrar essa diferença em vez de tratar todas as colunas como iguais. Os regimes de proteção de dados desse tipo costumam perguntar quais dados pessoais você mantém e quem pode vê-los, que é a mesma diferença; a interpretação cabe aos seus colegas jurídicos e de conformidade.

A **integridade** diz respeito a se os dados continuam corretos. Na maioria das tabelas, um erro pode ser corrigido por uma escrita posterior, mas `ledger_entries` e `audit_events` são append-only, e, no caso do ledger, a integridade importa mais do que o sigilo: um lançamento errado é corrigido por um novo lançamento corretivo por meio do `payments-service`, nunca por uma edição. Uma ameaça à integridade dessas duas tabelas é, portanto, diferente em natureza. É também aquela em que a evidência mais importa, já que um ledger que pode ser alterado em silêncio é um ledger em que ninguém pode confiar.

A **disponibilidade** diz respeito a se o banco de dados atende aos serviços que dependem dele. É a propriedade mais esquecida nas discussões de segurança, e aquela em que os acidentes predominam. Um bloqueio longo, uma consulta que varre uma tabela grande no primary ou uma enxurrada de conexões vinda de um loop de retry podem interromper a emissão de faturas e a cobrança de pagamentos tão completamente quanto um ataque. A disponibilidade recebe linhas de primeira classe no modelo, ligadas à política de conexão (Capítulo 5) e a onde as leituras podem ir (Capítulo 6).

Uma quarta propriedade fica ao lado dessas três: a **responsabilização**, ou seja, o fato de o registro poder dizer qual principal fez o quê. É a parte do Registro no acesso. No estado inicial, ela está ausente, já que o log mostra apenas o papel compartilhado, e essa ausência aumenta o custo de todas as outras ameaças.

Raio de impacto

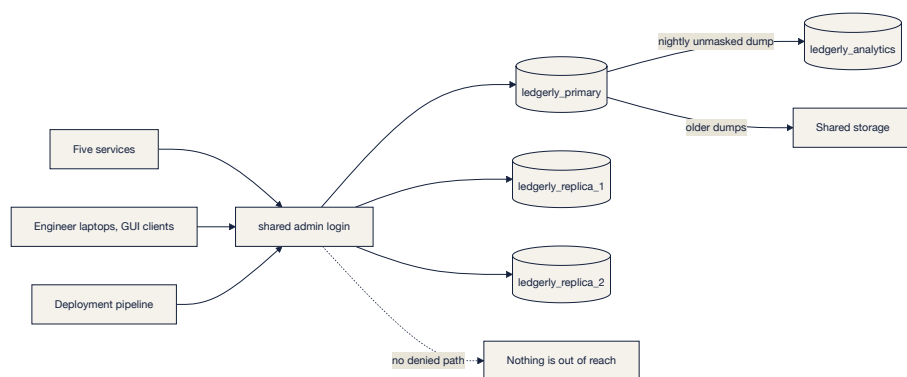
O modelo precisa de mais uma ferramenta para comparar ameaças: um modo de dizer até onde uma única falha se propaga. O **raio de impacto** é o quanto pode ser afetado se um principal, uma credencial ou uma ação der errado. O termo só é útil quando usado sobre algo específico, de modo que a pergunta é sempre “qual é o raio de impacto deste login?” e nunca “qual é o nosso raio de impacto?”

Um raio de impacto tem sete dimensões, e ajuda olhar cada uma separadamente:

- **Alcance dos dados:** em quais tabelas, colunas e linhas o principal pode tocar.

- **Alcance das ações:** se ele pode ler, escrever, excluir ou alterar o esquema.
- **Alcance dos ambientes:** a quais bancos de dados ele pode se conectar: o primary, as réplicas, a cópia de analytics.
- **Alcance no tempo:** por quanto tempo o acesso permanece válido, de uma sessão até para sempre.
- **Alcance das cópias:** onde mais os dados que ele pode ler foram duplicados, e quem consegue chegar a essas cópias.
- **Alcance das instruções:** quanto uma única instrução pode alterar antes que algo a interrompa: as linhas que ela toca, se roda em uma transação que pode sofrer rollback e se um timeout a limita.
- **Origem de rede:** quais redes e máquinas conseguem apresentar a credencial.

O login de administrador compartilhado, no estado inicial da Ledgerly, leva todas as dimensões ao máximo. Ele alcança todas as tabelas e todas as colunas, pode ler, escrever, excluir e alterar, conecta-se ao primary e às duas réplicas, não tem expiração e seus dados já têm cópias fora do primary. Qualquer instrução é ilimitada e faz commit imediatamente, e qualquer laptop na VPN pode apresentar a credencial. O diagrama mostra o que observar: tudo à esquerda guarda a mesma credencial, de modo que tudo à direita está dentro de um único raio.



Compare-o com o papel do estado-alvo `app_reporting_job`: somente leitura, em uma réplica, sem acesso às colunas Restricted. Um defeito que poderia expor dados Restricted no estado inicial é, para esse papel, uma consulta pesada em uma réplica. O defeito é idêntico; o raio de impacto mudou.

Para manter a comparação honesta, escreva o raio por login, como uma lista de perguntas e não como uma pontuação, e coloque dois logins lado a lado. O esboço abaixo é ilustrativo, um auxílio de revisão e não uma ferramenta em que o livro se apoie. Uma resposta desconhecida é registrada como desconhecida, porque um modelo que arredonda o desconhecido para “tudo bem” esconde exatamente os casos para os quais 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		

A comparação é útil justamente porque é desigual. Onde a segunda coluna diz “unknown”, o modelo encontrou uma pergunta que o desenho ainda não respondeu, e isso é um achado, não uma lacuna na tabela.

Aplicam-se dois limites. Um raio descreve o que um login poderia fazer, não o que ele vai fazer, de modo que é um limite de planejamento e não uma previsão. E ele não diz nada sobre a probabilidade de cada ameaça. Um raio estreito em uma porta aberta o dia todo ainda pode merecer mais atenção do que um raio largo em uma porta que se abre uma vez por trimestre, e é por isso que o modelo precisa combinar o raio com a exposição.

O modelo de ameaças da Ledgerly

Com os atores, as classes, as propriedades e o raio de impacto definidos, o modelo cabe em uma única tabela. A tabela abaixo cobre o estado inicial da Ledgerly. Cada linha é um ator e um caminho, com a classe de ameaça que mais importa para esse par, o impacto sobre as propriedades acima, os capítulos que fornecem o controle e o **risco residual**: o que continua verdadeiro depois que esses controles estão em vigor. “Exposição” é um julga-

mento qualitativo (alta, média ou baixa) de com que frequência a porta é usada e de quão fácil é fazer mau uso dela, e é escrita para que alguém possa discordar.

Ator	Caminho	Classe principal	Impacto	Expo-sição	Con-trola-do em	Risco residual
Serviços (os cinco)	Login de administrador compartilhado a partir da configuração de cada serviço	Erro, comprometimento	Qualquer tabela, todas as colunas; disponibilidade se as conexões se esgotarem	Alta	Capítulos 3, 4, 5	Um defeito ainda tem o alcance do seu próprio papel
reporting-job	Lê o primary em algumas execuções	Erro	Leituras pesadas competem com as escritas; disponibilidade do primary	Média	Capítulos 5, 6	Uma leitura pesada em uma réplica ainda deixa lentos os outros leitores da réplica
Engenheiros de on-call	Laptop, cliente GUI, senha de administrador compartilhada, auto-commit	Erro, comprometimento	Escritas sem escopo; integridade de qualquer tabela	Alta	Capítulos 7, 8, 10	Uma instrução digitada à mão dentro de um papel concedido ainda pode estar errada
Suporte	admin-console agindo sob o login compartilhado	Uso indevido, erro	Navegação por dados Confidential de clientes; edições no cliente errado	Média	Capítulos 3, 10, 12	O suporte ainda vê as colunas de que o caso de um cliente precisa
Equipe de dados	Restauração de analytics sem mascaramento e dumps antigos	Uso indevido, comprometimento	Confidencialidade de toda coluna Restricted e Confidential	Alta	Capítulos 12, 13	Dados mascarados ainda podem revelar valores por inferência
Fornecedor de contabilidade	Arquivos exportados	Erro, uso indevido	Confidencialidade das colunas exportadas e seu tempo de vida	Média	Capítulos 12, 13	O fornecedor guarda o arquivo depois da entrega

Ator	Caminho	Classe principal	Impacto	Exposição	Controlado em	Risco residual
Pipeline de deploy	Login de administrador armazenado como variável do pipeline	Erro, comprometimento	Integridade do esquema e dos dados; todas as tabelas	Média	Capítulos 4, 11	Uma migração revisada ainda roda com os direitos do proprietário
Equipe de plataforma e DBAs	Login de administrador compartilhado agora; <code>ledgerly_access_admin</code> no desenho-alvo	Erro, uso indevido, comprometimento	Grants, controles e evidências podem ser alterados	Média	Capítulos 3, 7, 14	O papel existe, então seu uso indevido é possível; resta apenas a detecção
Plano de controle (ativo)	Gateway, credential broker, armazenamento de auditoria	Erro, comprometimento	Todas as outras linhas mudam de significado	Média	Capítulos 4, 7, 14	O comprometimento de um componente enfraquece a evidência que ele guarda
Qualquer sessão humana	Uma instrução digitada à mão dentro de um papel concedido	Erro	Instrução ilimitada, com commit imediato	Alta	Capítulos 9, 10	Uma instrução pode ser válida, aprovada e ainda assim errada no efeito
Qualquer origem de rede	A VPN da empresa chega diretamente ao primary	Comprometimento	Uma credencial vazada funciona a partir de qualquer laptop na VPN	Média	Capítulo 7	Uma credencial roubada de uma origem permitida ainda funciona ali
Atacante externo	Qualquer credencial vazada das linhas acima	Comprometimento	Herda o raio do responsável pela credencial	Baixa a média	Capítulos 4, 7, 14	Uma credencial roubada dentro do seu tempo de vida funciona até expirar
Todos os atores	Nenhum login nomeado no log do banco de dados	Todas as classes	Responsabilização por todas as outras linhas	Alta	Capítulos 7, 14	O registro é tão confiável quanto o armazenamento que o guarda

Quatro observações decorrem da tabela, e são o motivo de construí-la.

Primeiro, o login de administrador compartilhado aparece na maioria das linhas. A tabela mostra um único problema com muitas faces, e o primeiro controle, um papel separado por principal (Capítulo 3), trata de uma boa parte dele de uma só vez. Um modelo que revela a ordem em que se deve trabalhar está cumprindo o seu papel.

Segundo, entre as maiores exposições estão portas que nunca foram pensadas para ser portas: a restauração de analytics e os dumps antigos. Nenhuma delas é um caminho de acesso no diagrama mental de alguém sobre o banco de dados. É por isso que os capítulos 6 e 13 tratam toda cópia de dados como uma porta com um responsável e uma expiração.

Terceiro, a coluna de risco residual é a parte honesta da tabela. Todo controle deixa algo: um papel concedido ainda pode ser usado de forma indevida, uma instrução válida ainda pode estar errada, um registro só pode ser considerado confiável até onde o seu armazenamento o for. Uma linha sem risco residual escrito é uma linha em que ninguém perguntou.

Quarto, as linhas do plano de controle, da sessão humana e da origem de rede nomeiam um ativo ou um caminho em vez de um ator, e a última linha também não nomeia um ator único. A falta de responsabilização é uma propriedade do sistema inteiro, e aumenta o custo de todas as outras linhas, porque sem um login nomeado nenhum incidente pode ter seu escopo definido. O Capítulo 14 fecha essa linha. Os capítulos anteriores constroem as identidades que tornam possível fechá-la.

Aviso: Um modelo de ameaças concluído é um mapa de onde a Ledgerly é mais fraca. Guarde-o em um lugar com as mesmas restrições de acesso dos sistemas que ele descreve, e não o cole em um ticket público nem em uma wiki de público amplo. Um documento que lista cópias sem mascaramento e credenciais compartilhadas é útil para quem o mantém e para mais ninguém.

Manter o modelo vivo

Um modelo de ameaças escrito uma vez é um retrato do momento, e a tabela acima fica desatualizada no dia em que a Ledgerly adiciona uma integração ou a equipe de dados ganha uma nova cópia. Trataria o modelo como um documento vivo com um responsável nomeado, como um runbo-

ok, mantido em um repositório com as mesmas restrições de leitura dos sistemas que ele descreve, já que a tabela lista fraquezas. Isso tem um custo. A configuração de acesso geralmente fica em um repositório com leitura mais ampla, de modo que uma mudança ali não aparece automaticamente na revisão do modelo. Aceitaria esse custo e fecharia a lacuna com uma regra: um pull request que altera a configuração de acesso inclui um link para a mudança correspondente no modelo, ou diz por que nenhuma é necessária.

Alguns eventos devem disparar uma atualização sem que ninguém precise lembrar:

- Um novo serviço, job, fornecedor ou ferramenta que alcança dados de produção.
- Uma nova cópia de dados: uma nova réplica, um export, uma atualização de staging, um dump para uma análise pontual.
- Uma mudança em quem consegue alcançar o quê: uma reorganização, um novo rodízio de on-call, o fim de um contrato com um fornecedor.
- Um incidente ou quase acidente, que é evidência sobre qual linha estava errada.
- Um achado em uma revisão de acessos (Capítulo 14), que é evidência sobre qual suposição se desviou.

Cada linha deve registrar o suficiente para que uma pessoa de fora possa contestá-la: o ator, o caminho, a classe, o impacto, o julgamento de exposição, o controle e o seu responsável, a data da última revisão e a suposição de que a linha depende. O último campo é o que as equipes pulam e o mais útil: “o export do fornecedor contém apenas estas colunas” é uma afirmação que pode ser verificada, e uma linha que depende de uma afirmação não verificada é um palpite.

Uma pequena verificação pode encontrar as linhas que ficaram obsoletas. Ela não decide nada. Aponta a uma pessoa as linhas com maior chance de estar erradas, e trata a ausência de data de revisão como obsolescência. Ela também recebe como entrada os eventos listados acima: quando um pipeline cria um login, um papel ou uma cópia de dados, ele registra um evento, e um evento que nenhuma linha cobre abre um item de revisão, de modo que a atualização não depende de ninguém se lembrar.

```

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

```

A cadência é um trade-off. Uma passada trimestral é algo que uma equipe pequena consegue manter, e uma anual raramente basta, mas escolheria o intervalo conforme a rapidez com que as portas da Ledgerly mudam. Os contatos de segurança e de conformidade definem as perguntas que a revisão faz, e a equipe de plataforma as responde com fatos sobre o sistema.

Trade-offs e custos

Um modelo de ameaças custa tempo para escrever e mais tempo para manter. Para uma equipe pequena, a primeira versão plausivelmente leva alguns dias de discussão cuidadosa, e a manutenção é um item recorrente que concorre com o trabalho de funcionalidades. Se ninguém for responsável por ele, o custo é pago uma vez e o benefício se deteriora. Em um sistema muito pequeno, com um serviço e um banco de dados, a tabela completa é mais do que o problema exige; meia página listando as portas e o raio de impacto no pior caso de cada uma basta.

O modelo também pode criar uma falsa confiança: uma tabela arrumada sugere que toda ameaça está nela. Ele lista as ameaças em que alguém pensou, e uma porta que não consta é invisível para ele. A melhor defesa são os gatilhos de atualização acima e o hábito de perguntar, depois de todo incidente: “qual linha teria previsto isto e, se nenhuma, qual é a linha que falta?”

Os julgamentos de exposição são subjetivos. Duas pessoas de engenharia avaliarão a mesma porta de forma diferente, e a discordância é informativa, mas uma avaliação implica mais precisão do que a evidência sustenta. Manteria isso em três níveis e resistiria a calcular pontuações.

Há um custo humano em listar grupos de pessoas como atores. Chamar as pessoas de on-call ou de suporte de atores de ameaça pode soar como uma acusação. A resposta está no enquadramento: o modelo descreve caminhos e papéis, não indivíduos, e suas entradas dizem o que o desenho permite. As equipes da tabela aparecem porque são as que têm a necessidade mais legítima de alcançar os dados.

Quando não usar o modelo completo: se ninguém consegue hoje responder “quem tem acesso?”, comece com um inventário de logins e cópias de dados antes de escrever ameaças, porque um modelo construído sobre um inventário impreciso descreve um sistema que não existe. E não o use como substituto de uma decisão. Ele classifica e explica; a equipe ainda escolhe quais controles construir primeiro e quais riscos aceitar por escrito.

Erros comuns

- **Modelar apenas o atacante externo.** Isso corresponde à imagem comum de segurança, e os controles são familiares. O custo é que erros e curiosidade ficam sem tratamento. Dê a cada uma das três classes as suas próprias linhas na tabela.
- **Listar controles em vez de ameaças.** “Temos uma VPN” registra uma resposta sem a pergunta. Diga primeiro o que pode dar errado e depois o controle, para que um controle possa ser aposentado quando a sua ameaça deixar de se aplicar.
- **Deixar de fora atores que não são pessoas.** Pipelines, jobs e exports são fáceis de ignorar porque ninguém faz login como eles, e tendem a deter o maior poder permanente. Liste todo principal e todo export.
- **Tratar os dados como uniformemente sensíveis.** Se toda coluna é “crítica”, o modelo não dá orientação sobre por onde começar. Use as quatro classes de dados (Restricted, Confidential, Internal, Public) para que o peso de uma ameaça acompanhe o peso dos dados.
- **Ignorar a disponibilidade.** Uma consulta descontrolada não parece um evento de segurança, então escapa das revisões de segurança. Para

um negócio de pagamentos, ela pode importar tanto quanto. Dê à disponibilidade as suas próprias linhas.

- **Escrevê-lo uma única vez.** O modelo é preciso por algumas semanas e depois vira folclore. Atribua um responsável e adicione gatilhos de atualização.

Pontos-chave

- Um modelo de ameaças é uma lista estruturada e curta de atores, caminhos, classes de ameaça, impactos e controles, e a sua função é permitir que uma equipe compare as portas entre si.
- Erros, uso indevido e comprometimento exigem controles diferentes, e um desenho que considera apenas quem está de fora deixa as duas primeiras classes sem tratamento; aprovar a intenção trata do uso indevido, enquanto aprovar a instrução e o seu efeito esperado trata dos erros.
- Todo ator pertence à lista, inclusive pipelines, jobs em lote, exports de fornecedores e a equipe de plataforma que opera os controles, porque os principais como os quais ninguém faz login muitas vezes detêm o maior poder permanente, e o plano de controle também é um ativo.
- Confidencialidade, integridade e disponibilidade precisam, cada uma, das suas próprias linhas, e a responsabilização, a capacidade de dizer qual principal fez o quê, aumenta o custo de todas as outras ameaças quando está ausente.
- O raio de impacto sempre diz respeito a um login, uma credencial ou uma ação específica, e é descrito em sete dimensões: dados, ação, ambiente, tempo, cópias, instrução e origem de rede.
- No estado inicial da Ledgerly, um único login de administrador compartilhado aparece em quase todas as linhas, de modo que um papel separado para cada principal é o controle que reduz a maior parte da tabela de uma só vez.
- O modelo só é tão bom quanto a sua manutenção: dê a ele um responsável, ligue as atualizações a eventos como novos serviços e novas cópias de dados e mantenha-o onde somente as pessoas que mantêm o sistema possam lê-lo.

O modelo mostra quais portas mais importam, e, no caso da Ledgerly, quase todas levam de volta a um único login. O Capítulo 3 começa a desmontar esse login: quem são os principais, o que um papel deve conter e quanto pouco cada um deles precisa.

Capítulo 3 — Identidades, papéis e privilégio mínimo

Imagine uma quinta-feira à noite na Ledgerly. Uma nova tabela, `invoice_disputes`, entrou no ar na terça-feira, e o contato de segurança faz à equipe de plataforma uma pergunta curta e razoável: quem pode lê-la? Alex começa pelos grants e não encontra nenhum, porque todo chamador se conecta como `admin`, que é proprietário da tabela e pode ler qualquer coisa. Alex então procura quem de fato a leu. O log do banco de dados tem milhares de linhas sobre a tabela, todas escritas pelo mesmo login: o worker, o relatório noturno, dois engenheiros em seus laptops e a ferramenta de suporte.

Ninguém consegue situar a nova tabela. A resposta honesta a “quem pode lê-la?” é “quem puder chegar ao banco de dados”, e a resposta honesta a “quem leu?” é “alguém”. Ninguém fez nada de errado. O desenho nunca pediu ao banco de dados que distinguísse esses chamadores, nem que limitasse qualquer um deles.

O Capítulo 1 dividiu o acesso em cinco partes: **Identidade**, **Grant**, **Caminho**, **Execução** e **Registro**. A tese do livro é que o acesso é um problema de autorização, execução e responsabilização, e não de distribuição de credenciais. Este capítulo serve ao pilar da autorização. Ele cobre as duas primeiras partes, quem está pedindo (Identidade) e o que esse principal pode fazer (Grant), e marca onde os grants terminam, porque é ali que a parte da Execução assume (Capítulo 9).

Ele acrescenta os primeiros passos do estado-alvo do exemplo corrente do livro. Toda tabela pertence ao `ledgerly_owner`, e ninguém se conecta como proprietário. Cada serviço recebe o seu próprio papel de aplicação, com os grants mais estreitos que funcionem. O capítulo também apresenta o esquema completo de papéis, o modo como os principais chegam aos papéis, os grants em nível de coluna e o teto para o acesso just-in-time. Os capítulos seguintes se apoiam nessas definições e não as repetem.

O problema e por que ele é difícil

Um banco de dados responde a duas perguntas a cada instrução: quem é este, e esse principal tem permissão para fazer isto? Uma conta compartilhada torna a primeira resposta sem sentido, e uma conta ampla faz a segunda resposta ser sempre sim. É fácil chegar a ambas. Um login, uma senha e todos os direitos é a coisa mais barata que funciona no dia em que uma equipe cria o seu primeiro banco de dados.

O custo chega depois, como evidência ausente. Quando as pessoas e os serviços que compartilham o login de administrador (digamos, vinte pessoas, cinco serviços, o pipeline de deploy e a etapa de export) o usam todos, o log do banco de dados não consegue dizer qual deles executou uma instrução. O log da aplicação descreve o que a aplicação pretendia fazer, não o que chegou ao banco de dados. Esse é um problema de design, não de disciplina, porque nenhum cuidado individual traz de volta a informação que o desenho descartou.

O remédio óbvio é escrever uma política: conceder somente o que for necessário. Uma política não funciona aqui, porque o banco de dados torna fácil a escolha ampla. Conceder tudo exige uma instrução e nunca falha. Conceder o menor conjunto que funciona exige conhecer cada consulta que um workload executa, inclusive o job mensal de que ninguém se lembra e o caminho de retry que roda duas vezes por ano. Um grant estreito demais derruba a produção no pior momento. Um grant largo demais não derruba nada, então ninguém o percebe. Sob essa assimetria, os privilégios só fazem crescer.

A propriedade acrescenta uma segunda armadilha. Na maioria dos motores, o proprietário de uma tabela pode alterar sua estrutura, removê-la e conceder acesso a outros. Se um serviço se conecta como proprietário porque cria tabelas na inicialização, o serviço detém o poder de reescrever o modelo de acesso do banco de dados. Se uma pessoa é a proprietária, o modelo depende do vínculo empregatício dessa pessoa. A propriedade é um privilégio estrutural, e não deve ficar em nada que lide com tráfego nem em nada que lide com uma carreira.

O remédio é uma estrutura, não uma instrução: um login distinto por principal, um pequeno conjunto de papéis que carregam privilégios e um jeito de manter ambos em sincronia com o que a equipe pretende.

Principals, logins e papéis

Este capítulo trabalha com Identidade e Grant, e toca no Caminho onde um papel diz de onde pode se conectar. Um **principal** é qualquer coisa que se autentica. No mundo da Ledgerly, há quatro tipos que vale separar.

- **Humanos.** Engenheiros de backend de on-call, equipe de suporte, equipe de dados e equipe de plataforma. Eles vão e vêm, mudam de equipe e trabalham de lugares diferentes.
- **Serviços.** `billing-api`, `invoice-worker` e `payments-service`. Têm execução longa, rodam em muitas instâncias e carregam o tráfego mais pesado.
- **Jobs.** `reporting-job` e a sua etapa de export. Rodam de acordo com um agendamento e terminam. Um job costuma precisar de menos do que um serviço, e por menos tempo.
- **Ferramentas.** O `admin-console` é o caso mais claro. É um serviço na forma como se conecta, mas age em nome de uma pessoa, de modo que dois principals estão envolvidos em cada ação: a ferramenta e o humano por trás dela.

O fornecedor externo de contabilidade nunca se conecta a nenhum banco de dados da Ledgerly; ele recebe arquivos, e o Capítulo 13 trata desse caminho.

Identidade, login e papel são três coisas diferentes

O livro mantém três palavras separadas, porque misturá-las é a origem da maior parte da confusão no desenho de acesso.

Uma **identidade** é a afirmação verificada de que um principal é quem diz ser. Um **login** é a conta em nível de banco de dados com a qual um principal se conecta. Um **papel** é um pacote nomeado de privilégios. Um principal tem uma identidade, conecta-se por meio de um login, e o login é membro de um ou mais papéis. O login é o que aparece no log do banco de dados. O papel é o que decide o que o login pode fazer.

Isso dá ao modelo uma frase que vale lembrar: ninguém faz login como um papel. Um papel existe para ser concedido a logins e para guardar privilégios, e nunca é uma pessoa. Se o log mostra o nome de um papel onde

deveria haver uma pessoa ou uma instância de workload, o modelo foi contornado em algum ponto.

Para humanos, o access gateway (Capítulo 7) cria um login por sessão que carrega a identidade da pessoa, e o login recebe `human_readonly` por padrão. Para serviços, cada instância se conecta com um login que é membro do seu papel `app_*`. No estado-alvo, esse login é dinâmico e de curta duração (Capítulo 4). No estado inicial, é o login de administrador compartilhado. A consequência para a pergunta central é simples: o log nomeia um login, e um login corresponde a uma pessoa ou a uma instância de workload, de modo que “quem executou essa instrução?” tem resposta.

Por que uma conta compartilhada destrói a evidência

O Capítulo 1 apresentou o argumento completo; a forma resumida é que uma conta compartilhada não consegue atribuir uma instrução, não pode ser revogada para uma só pessoa e não pode ter um escopo menor do que o que o seu chamador mais amplo precisa.

Um caminho exige cuidado: o `app_admin_console` é um workload, mas seus usuários são pessoas. O console repassa a identidade do principal humano a cada ação, e o esboço abaixo mostra como.

```
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 })
```

O login do banco de dados nomeia o workload, e o contexto da sessão e o `audit_events` carregam a pessoa e o correlation id, de modo que o log da aplicação, a trilha de auditoria e o próprio registro da instrução comparti-

lham uma mesma chave. O Capítulo 14 trata de como os registros são unidos. O contexto é definido pelo console, portanto é uma afirmação feita pelo workload e não um fato verificado pelo banco de dados; a ressalva sobre o pooler na seção sobre regras de linha também se aplica a ele.

Uma hierarquia de papéis em quatro camadas

Construiria o esquema de papéis em quatro camadas, da que ninguém usa para fora, até as que as pessoas usam. Cada camada tem uma função.

Primeira camada: o proprietário

O `ledgerly_owner` é proprietário de todo objeto de esquema: esquemas, tabelas, views e sequences. Ninguém faz login com ele. Ele existe para que a propriedade não fique ligada nem a uma pessoa nem a um serviço.

```
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
```

Segunda camada: o migrator

O `ledgerly_migrator` executa as mudanças de esquema. O pipeline de deploy usa o seu login, os humanos não, e ele não detém nenhum privilégio de dados próprio. Ele é membro do `ledgerly_owner`. Em alguns motores, essa participação transmite os privilégios do proprietário sem nenhuma etapa explícita, mas não a propriedade: um novo objeto pertence ao papel que estiver ativo quando ele é criado, de modo que a migração sempre assume o `ledgerly_owner` de forma explícita.

O migrator detém privilégios permanentes de proprietário, exercidos por meio de uma credencial de curta duração. Os privilégios nunca expiram; apenas cada credencial expira. Três coisas o limitam: a credencial vive apenas no pipeline, é de curta duração (capítulos 4 e 11) e o procedimento do pipeline é a única maneira autorizada de usá-la. As mudanças de esquema são possíveis sem que nenhuma pessoa ou serviço detenha a pro-

priedade e, se esses limites se mantiverem, cada uso do proprietário corresponde a uma execução do pipeline. O Capítulo 11 trata de como as migrações são revisadas.

Os backfills que pertencem a uma mudança de esquema rodam dentro da migração como `ledgerly_migrator`. Qualquer outra alteração em massa de dados é uma correção de dados e segue o Capítulo 10. Traço essa linha porque o poder do migrator é grande, e ele não deve se tornar o caminho para toda mudança incômoda.

Terceira camada: papéis de aplicação

Cada workload recebe o seu próprio papel. Um papel por serviço custa cinco definições para manter em sincronia com cinco bases de código. Em troca, o raio de impacto de cada credencial tem o tamanho do trabalho daquele serviço: se a do `invoice-worker` vazar, ela expõe o que o worker podia fazer, e o worker nunca precisou do `payment_tokens`.

Os papéis são `app_billing_api`, `app_invoice_worker`, `app_payments_service`, `app_reporting_job` e `app_admin_console`. O prefixo diz quem o usa: `app_` para workloads, `human_` para pessoas. Os papéis não carregam nomes de pessoas nem o nome de um ambiente.

Quarta camada: papéis humanos

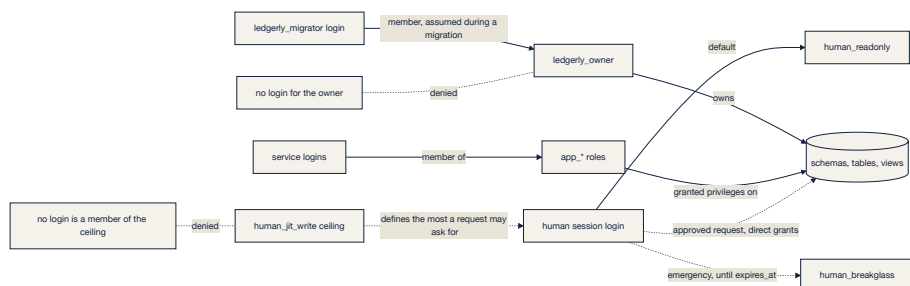
As pessoas recebem três papéis, e eles diferem na forma como são mantidos.

- `human_readonly` é o padrão. Lê para fins de investigação, retém as colunas Restricted e mascara as colunas Confidential onde o Capítulo 12 determina.
- `human_jit_write` é um teto. Nunca é concedido a um login. Cada requisição aprovada concede um subconjunto dele, em tabelas e colunas nomeadas, diretamente ao login da sessão, e depois o retira.
- `human_breakglass` é acesso amplo de emergência, com logging pesado (o Capítulo 8 o define) e revisão obrigatória depois.

O teto é o menos intuitivo dos três, e volto a ele depois da mecânica dos grants.

O diagrama

O diagrama abaixo mostra quem é membro de quê e quem é proprietário de quê. Observe as duas coisas que não estão conectadas: não existe nenhum login para o `ledgerly_owner`, e nenhum login é membro do `human_jit_write`.



Grants: esquema, tabela e coluna

Um **grant** é o ato, e o registro resultante, de dar um privilégio a um papel, dar um papel a um login ou a outro papel ou, apenas no caso do acesso JIT, dar um privilégio diretamente a um login de sessão. Um **privilégio** é a coisa específica que um grant permite, como `SELECT` em uma tabela. Os três significados mantêm o vocabulário preciso, e neste capítulo direi a qual deles me refiro.

Comece do nada

O ponto de partida de qualquer papel é nenhum privilégio. Adicione grants até o workload funcionar, e pare. Os motores diferem quanto ao que um novo login pode fazer por padrão: em alguns, qualquer login pode ver nomes de objetos ou criar objetos em um esquema padrão. Liste o que todo login recebe automaticamente e remova o que um papel não precisa.

Os privilégios se aplicam em três níveis e se somam: um papel detém a união de tudo o que lhe foi concedido em qualquer nível, e um grant estreito nunca retira um grant amplo.

- **Nível de esquema.** Se um papel pode ver os objetos de um esquema. Nos motores que o exigem, os grants em nível de tabela não fazem nada sem ele, o que torna o acesso ao esquema uma chave geral útil.

- **Nível de tabela.** `SELECT`, `INSERT`, `UPDATE` e `DELETE` em uma tabela inteira.
- **Nível de coluna.** Os mesmos privilégios, apenas em colunas nomeadas.

Os grants em nível de coluna importam porque os dados sensíveis estão distribuídos de forma desigual: `customers` guarda dados de contato Confidential, `bank_accounts` guarda `iban` e `account_holder_name`, que são Restricted, e `payment_tokens` guarda `token_value`, que é Restricted. Um grant em nível de tabela diz sim a toda coluna da tabela. Como os privilégios se somam, restringir colunas significa conceder apenas em nível de coluna e não manter nenhum grant em nível de tabela naquela tabela para o papel, direto ou por meio de outro papel de que ele faça parte.

Grants de aplicação

Os dois primeiros papéis de aplicação, em nível de tabela:

```
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
```

O `app_billing_api` lê `ledger_entries` e `payments`, mas não pode escrevê-las, porque somente o `payments-service` escreve em `payments`, `bank_accounts` e `payment_tokens` e insere em `ledger_entries`. Nenhum dos papéis detém `DELETE` em nada; se um serviço precisar dele, o grant é adicionado para aquela tabela com uma justificativa, e a revisão percebe.

Grants em nível de coluna para os outros papéis

Os papéis com uma adaga na matriz têm restrição de colunas. As listas de colunas abaixo são ilustrativas: quais colunas Confidential aparecem, e em que forma mascarada, é decisão do 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
```

O último grant é o de que mais gosto. O `app_admin_console` pode alterar o `status` de uma fatura e nada mais na linha, de modo que o grant declara a intenção e o banco de dados a impõe mesmo que o código da ferramenta mude depois. O `app_reporting_job` não detém nenhum privilégio de escrita em nenhuma tabela principal: ele grava seus resultados em `ledgerly_analytics`.

Nota: Nos motores que replicam fisicamente, os papéis são definidos uma única vez para o cluster inteiro, de modo que o lugar de onde cada papel pode se conectar é uma regra da camada de conexão (capítulos 5 e 7) e não algo que os grants expressem. O Capítulo 6 traz os detalhes.

A matriz como referência

A tabela abaixo é a referência do estado-alvo para os seis papéis que leem ou escrevem nas tabelas principais. S é `SELECT`, I é `INSERT`, U é `UPDATE`, D é `DELETE`, um traço significa nenhum acesso e uma adaga (†) significa restrição de colunas: as colunas Restricted são retidas e as colunas Confidential são mascaradas onde o Capítulo 12 determina.

Tabela	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†

Tabela	app_ billing_ api	app_ invoice_ worker	app_ payments_ service	app_ reporting_ job	app_admin_ console	human_ readonly
payments	S	—	S, I, U	S†	S†	S†
invoices	S, I, U	S, I, U	S	S	S, U (so- mente 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

Leio a matriz por coluna. Cada coluna é a descrição de cargo de um workload escrita como privilégios, e quem revisa pode perguntar se ela continua verdadeira. Ao longo dela, apenas o `payments-service` alcança `bank_accounts`, somente o `payments-service` toca em `payment_tokens`, exceto por uma leitura com restrição de colunas do `human_readonly`, e nenhum papel humano escreve por padrão.

Append-only é estrutural

Nenhum papel de aplicação, nenhum papel humano e nenhum grant JIT detém `UPDATE` ou `DELETE` em `ledger_entries` ou `audit_events`, e isso inclui o `human_breakglass`. Uma correção é um novo lançamento, nunca uma edição. As correções do ledger são emitidas apenas pelo caminho de correção do `payments-service`, nunca por um login humano.

Há uma exceção estrutural: o `ledgerly_owner`, e portanto uma migração em execução, pode alterar essas tabelas, porque o proprietário pode alterar ou remover a tabela e conceder a si mesmo qualquer privilégio sobre ela. É por isso que o Capítulo 11 revisa toda migração que toca nelas. O controle está na revisão, não no privilégio.

Padrões, propriedade e o superusuário

Default privileges

Muitos motores permitem que um grant se aplique a tabelas criadas depois, seja como um default privilege ligado a um proprietário, seja como um grant para o esquema inteiro. Isso é conveniente no início e uma armadilha depois, porque uma nova tabela herda em silêncio o que o grant disser.

Minha preferência é um padrão vazio. Uma nova tabela é legível por ninguém além do proprietário, e a migração que a cria declara cada grant no mesmo arquivo. O custo são algumas linhas a mais por migração; o benefício é que uma única revisão mostra todo o efeito de acesso de uma mudança.

Aviso: Um default privilege que concede SELECT em tabelas futuras a um papel amplo, como o `human_readonly`, expõe a próxima tabela sensível no instante em que ela é criada. Revise esses padrões primeiro e remova os que cobrem tabelas futuras.

A notação é ilustrativa, porque os default privileges não têm uma sintaxe 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;
```

O perigo da propriedade

Nos motores comuns, o proprietário pode remover um objeto, alterar sua definição, conceder privilégios a qualquer pessoa e, em muitos casos, ignorar restrições em nível de linha que se aplicam a outros. Quem detém a propriedade pode alterar o próprio modelo de acesso.

Por isso as regras em torno do `ledgerly_owner` são estritas, e o rigor é barato. Não existe nenhum login para ele. Os papéis de aplicação nunca são proprietários de tabelas, nem mesmo daquelas que criam para uso pró-

prio, porque “o serviço cria a sua tabela na inicialização” é como um workload acaba sendo proprietário de parte do esquema. O proprietário é o papel menos usado do sistema, que é o estado correto.

O superusuário

A maioria dos motores tem um papel administrativo embutido que ignora as verificações de privilégio, embora os serviços gerenciados possam retê-lo. Aqui chamo esse papel de superusuário; ele fica fora do esquema de papéis da Ledgerly porque vem com o motor.

Trataria esse papel como um instrumento selado. Nenhuma aplicação, job ou sessão humana o usa no dia a dia. Sua credencial fica guardada onde a equipe de plataforma pode recuperá-la e a recuperação deixa um registro, e ele serve para operações em nível de motor que nenhum papel consegue fazer, como reparar a replicação. O limite honesto é que ele pode desfazer todo controle deste capítulo, e não conheço um desenho que remova esse poder sem tornar a recuperação impossível. Torne o seu uso raro, visível e revisado; o Capítulo 15 volta a ele. A administração de rotina, como aplicar definições de papéis ou emitir logins, usa o `ledgerly_access_admin`. Se o superusuário é usado para tarefas de rotina, falta a algum papel um privilégio que ele deveria ter.

O `ledgerly_access_admin` é um papel usado por três principais, cada um por meio do seu próprio login: a etapa do pipeline que aplica a definição de acesso, o credential broker (Capítulo 4) e o access gateway (Capítulo 7). Ele cria logins e gerencia participações, não é proprietário de nenhuma tabela e não detém nenhum privilégio de dados, e mesmo assim é um poder amplo: quem o detém pode conceder qualquer coisa dentro dos papéis que administra. Logins separados mantêm os três distintos no log, embora não no que podem fazer.

O que limita o poder do próprio plano de controle, portanto, não é o papel, mas o entorno. Cada um dos três logins só é alcançável a partir do seu próprio componente. Toda ação vai para o log de auditoria (Capítulo 14), e a verificação de drift abaixo compara o resultado com o código. As mudanças na definição de acesso passam por um pequeno conjunto próprio de revisores. Esses limites tornam o uso indevido visível; não o tornam impossível, e diria isso na revisão de acessos.

Estreitar ainda mais: views, regras de linha e tenants

Views

Uma **view** é uma consulta armazenada que se comporta como uma tabela. Um papel com acesso à view lê o que a consulta expõe sem deter um privilégio sobre a tabela subjacente.

```
// 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: Em muitos motores, uma view roda com os privilégios do seu proprietário, não do chamador. É isso que faz o estreitamento funcionar, e é também por isso que uma view que seleciona demais vaza esse excesso para todo papel que tenha grant sobre a view. Alguns motores também permitem escolher o outro comportamento. Verifique o que o seu faz por padrão e teste a view com um papel que não detenha nenhum privilégio sobre a tabela base.

Prefiro grants em nível de coluna onde o motor os oferece, porque o grant fica junto da tabela. Uma view é melhor quando a forma é o ponto central: um join, um valor calculado ou uma interface estável sobre uma tabela que muda. O livro não usa uma view por requisição para o acesso JIT, por dois motivos: uma view precisa ter um proprietário, então o gateway que a cria precisaria de poder de proprietário, e uma view feita para cada requisição quebra o modelo de propriedade.

Regras de linha e tenants: uma regra de decisão

Uma regra de linha estreita as linhas dentro de uma célula da matriz e não muda o que a célula diz. O suporte olha um cliente por vez, então uma regra como “o `app_admin_console` vê apenas as linhas de `invoices` em que `customer_id` é igual ao cliente do caso” se encaixa onde um filtro de colunas não se encaixa. Os motores divergem na sintaxe, então o livro não mostra nenhuma notação.

Os clientes da Ledgerly são empresas, e cada uma espera não ver as faturas de outra. Esse isolamento é, na maior parte, trabalho da aplicação: o `billing-api` filtra por cliente em cada consulta, e nenhum papel separa um tenant de outro, porque o papel de aplicação precisa ler todos eles para atender a qualquer um.

Use uma regra de linha, para o suporte ou como uma segunda camada de tenant, somente quando duas condições se cumprirem: uma linha errada é grave, e a aplicação define o valor de contexto de forma confiável para toda sessão. O custo é que toda instrução carrega o predicado e algo precisa definir o contexto por sessão. Atrás de um pooler, um valor definido em uma sessão pode chegar a quem pegar a mesma conexão em seguida (o Capítulo 5 volta a isso). Quando o contexto está ausente, a regra deve retornar nenhuma linha, e o primeiro teste é esse caso. Uma regra que falha aberto é pior do que nenhuma. Bancos de dados separados são uma fronteira mais forte a um preço operacional muito maior; qual se encaixa depende de quantos tenants você tem e do que prometeu a eles.

O teto para o acesso just-in-time

O `human_jit_write` define o teto: o máximo que uma requisição just-in-time (JIT) pode pedir, expresso como privilégios em tabelas e colunas nomeadas. O acesso JIT é o acesso concedido para uma tarefa específica e por um tempo limitado; o Capítulo 8 trata de aprovações e tempos de vida, e este capítulo apenas define o teto.

O papel nunca é concedido a um login. Uma requisição é uma lista de triplas (privilégio, tabela, colunas) que cabe dentro do teto, armazenada com a requisição. Na aprovação, exatamente essas triplas são concedidas diretamente ao login de sessão do solicitante, e na expiração exatamente essas triplas são revogadas. Um papel compartilhado não pode ter escopo por requisição, então o escopo por requisição fica nos grants e o papel traça a borda externa.

```

// 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

```

Leia o bloco pelo que ele deixa de fora. `bank_accounts` e `payment_tokens` ficam abertas apenas para as poucas colunas nomeadas que não são Restricted, e nunca para identificadores de linha ou timestamps: `iban`, `account_holder_name` e `token_value` nunca podem aparecer em uma requisição JIT. O INSERT se limita às quatro primeiras tabelas e não pode nomear `id` nem `created_at`, de modo que uma linha restaurada ou nova sempre recebe valores atribuídos pelo banco de dados. As duas tabelas append-only e os registros de requisições de acesso, que pertencem ao gateway, ficam inteiramente fora do teto. Toda tripla, exceto um DELETE, nomeia as suas colunas.

Dois custos são declarados uma única vez, aqui. As colunas que ligam uma linha ao seu proprietário, como `customer_id`, não são Restricted, de modo que um UPDATE JIT pode alterá-las. Essa mudança de vínculo de proprietário permanece dentro do teto, mas é uma classe de operação pe-

rigosa: a política de execução a barra com um gate e a encaminha ao caminho de produto (Capítulo 10), e ela não roda como SQL bruto. O gate e o caminho de produto são o controle, não a aprovação four-eyes isolada. E um SELECT JIT mostra sem mascaramento qualquer coluna Confidential que nomeie, o que faz parte do que quem aprova aceita.

O que mantém o teto honesto é uma verificação, não a confiança. O gateway recusa qualquer requisição fora dele (o Capítulo 8 mostra a verificação), e o teste de drift abaixo afirma que nenhum login é membro do `human_jit_write`, a falha que transformaria um teto em um grant permanente.

Onde o motor não tem grants em nível de coluna

Nem todo motor oferece grants em nível de coluna. O livro usa exatamente um fallback: onde o motor não consegue conceder em colunas, as colunas Restricted passam para tabelas separadas, `bank_account_secrets` e `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
```

Com o fallback, as tabelas principais não têm nenhuma coluna Restricted, de modo que um grant JIT em nível de tabela sobre elas não consegue alcançar dados Restricted. O custo: o detalhe da requisição cai para o nível de tabela, e quem aprova passou a aprovar toda coluna não Restricted da tabela, inclusive as Confidential, e não apenas as que o solicitante nomeou.

Break-glass e as operações que ficam acima do esquema

O `human_breakglass` é o único papel humano que lê colunas `Restricted`. Ele lê todas as tabelas e escreve `INSERT` e `UPDATE` em todas as tabelas, exceto nas duas `append-only`; em `bank_accounts` e `payment_tokens` ele escreve apenas `UPDATE`, e somente em colunas não `Restricted`. Serve para emergências, não para correções de rotina, e está ausente da matriz porque somente uma sessão o detém, por uma janela curta; nenhum `workload` permanente o detém.

As colunas `Restricted` só são alteradas por meio de um caminho de aplicação. Um humano nunca as escreve com SQL bruto, em uma sessão `JIT` ou de outra forma: o `admin-console` pede ao `payments-service`, o único que escreve, depois que uma segunda pessoa aprova, e a mudança é registrada em `audit_events`. O Capítulo 10 descreve esse caminho. Esses fatos marcam a borda do que os `grants` conseguirão fazer. Um `grant` diz em quais tabelas e colunas uma sessão pode tocar; não consegue dizer qual instrução, quantas linhas, nem se esta mudança específica é a que foi aprovada. Esse é o trabalho da parte da Execução, e o Capítulo 9 trata de como o gateway verifica cada instrução. Saber onde os `grants` terminam ajuda a decidir a que lugar pertence uma nova capacidade.

Papéis como código e drift

Um esquema de papéis que existe apenas no banco de dados é um que ninguém pode revisar. Os `grants` acima são curtos o bastante para serem lidos, e devem ficar no controle de versão, mudar por revisão e ser aplicados pelo pipeline, como o esquema que protege.

Onde ficam as definições

Manteria dois tipos de arquivo. As migrações criam tabelas e declaram os `grants` de cada tabela. Uma definição de acesso separada e revisada lista os papéis, as participações entre papéis e o teto. O pipeline aplica ambos em etapas separadas. A definição de acesso é aplicada por meio do login próprio da etapa do pipeline no `ledgerly_access_admin`, o mesmo papel que o broker e o gateway usam por meio dos seus próprios logins; ele não detém nenhum privilégio de dados. Criar papéis é um poder mais forte do

que alterar tabelas, então a sua etapa deve ter um pequeno conjunto próprio de revisores. A pergunta de quem revisa não é se o SQL é válido, mas se este grant é o mais estreito que faz a mudança funcionar. Quem revisa, com a matriz aberta, consegue responder isso em um minuto.

Estreitar um sistema existente

A Ledgerly começa com um único login compartilhado que é proprietário de tudo. Não passaria disso para a matriz em uma única mudança, porque um grant estreito demais derruba a produção e a equipe não tem como saber o que deixou passar até que algo falhe. Um caminho em quatro passos:

1. **Dê a cada workload o seu próprio login primeiro, com os mesmos privilégios de antes.** Nada quebra, e o log passa imediatamente a nomear o workload.
2. **Observe.** Use as instruções registradas sob cada login para listar o que o workload de fato toca, ao longo de um período longo o bastante para incluir o lento job mensal. Um log de instruções pode conter valores literais, então trate-o como sensível: ele pode conter dados de clientes ou valores Restricted que foram colados em uma consulta. Restrinja quem o lê, defina a sua retenção e prefira registrar o formato das instruções com parâmetros onde o motor oferecer.
3. **Defina o papel estreito e mova o login para ele,** primeiro em um ambiente de staging que roda os mesmos jobs, depois em produção, mantendo os privilégios antigos à mão por um curto período.
4. **Remova os privilégios antigos** e fique atento a instruções negadas.

O custo é tempo de calendário, não esforço. As equipes tendem a pular o segundo passo, e o resultado previsível é um caminho raramente usado que quebra meses depois.

Drift

O **drift** é a diferença entre o acesso que o código declara e o acesso que o banco de dados detém. Ele aparece quando um problema é consertado à noite com um `GRANT` escrito à mão: o grant funciona, o incidente termina

e ninguém registra por escrito. A verificação compara o declarado com o real em um agendamento e deve falhar de forma ruidosa quando não consegue ler um 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)` é `human_readonly` para um login de sessão do gateway, também `human_breakglass` enquanto uma requisição break-glass estiver aberta, e o único papel `app_*` para o qual o broker emitiu o login. A verificação, portanto, detecta um `GRANT app_payments_service T0 <session login>` ou um `GRANT ledgerly_owner T0 <login>` escrito à mão, assim como um grant direto de privilégio.

Aqui, `roleGrants` cobre apenas as participações entre papéis e os grants de privilégio a papéis, que o código declara. Os logins não estão no código: eles vêm e vão com as sessões. Os logins humanos são verificados em relação aos registros do gateway. Os grants diretos de todo login, inclusive os logins de serviço, são comparados com as triplas de `access_request_privilege` que ainda não expiraram. Um login de serviço não tem requisições, então qualquer grant direto nele é sinalizado; é assim que um `GRANT` de privilégio escrito à mão a um login de serviço durante a noite é percebido. Um grant que sobreviveu à sua requisição, ou um membro de break-glass sem requisição aberta, é sinalizado mesmo quando o job de revogação informa sucesso. O acesso JIT funciona por grants diretos e a expiração por um job de revogação; se o job parar, os grants permanecem, e esta verificação percebe isso independentemente do job (o Capítulo 8 trata do par). A verificação lê apenas grants e participações. Ela não vê o que uma sessão faz com eles, o que de novo é a parte da Execução.

A verificação não repara nada, porque uma revogação automática de grants desconhecidos em produção é uma nova maneira de derrubar a produção à noite. Tampouco diz se o estado declarado está certo: um grant errado, corretamente declarado, passa. Esse julgamento cabe à revisão de acessos do Capítulo 14.

Trade-offs e custos

Cada controle deste capítulo tem um preço, e o preço é pago por pessoas específicas.

Papéis por workload custam manutenção. Cinco papéis significam cinco definições para manter em sincronia com cinco bases de código. Quando um serviço precisa de uma nova tabela, alguém atualiza o papel na mesma mudança; se o grant chegar depois do código, o deploy falha. A mitigação é tornar o grant parte da migração. Com um serviço e duas pessoas de engenharia, esse trabalho extra não é nada desprezível, e um esquema menor, com um proprietário, um papel de aplicação e um papel somente leitura, é um primeiro passo razoável.

Grants estreitos causarão instruções negadas. Algumas negações são erros nos grants e outras são código que foi mais longe do que deveria. Quem está de on-call paga por ambas quando uma negação aparece durante um incidente. Rodar o staging com os mesmos papéis da produção reduz a surpresa; ignorar o custo leva as equipes a ampliar um grant às pressas, e a pressa é como o drift começa.

Grants em nível de coluna aumentam a carga de revisão. Cada lista de colunas precisa ser mantida quando a tabela muda. Uma nova coluna sensível fica exposta por padrão sob um grant em nível de tabela, então, para tabelas sensíveis, concedo em colunas nomeadas e aceito as linhas extras. Onde o motor não tem o recurso, o fallback custa joins, tabelas extras para migrar e uma aprovação JIT mais grosseira.

Views e regras de linha acrescentam peças móveis. Valem o que pesam quando a forma ou o isolamento é o ponto central, e são uma troca ruim quando adicionadas para dar a sensação de cuidado. Uma regra de linha cujo contexto pode desaparecer é teatro de segurança: acrescenta atrito a toda consulta e falha aberto justamente no caso para o qual foi escrita.

A disciplina de propriedade desacelera algumas pessoas. Engenheiros acostumados a criar uma tabela em um console agora precisam passar por uma migração. O atrito é intencional, e vale menos em um banco de dados de desenvolvimento descartável do que em produção. Aplique o modelo primeiro à produção e o relaxe em outros lugares somente quando conseguir dizer o que está abrindo mão.

Quando não usar tanta estrutura. Um protótipo com uma pessoa desenvolvendo e sem dados de clientes não precisa do esquema de papéis completo. A divisão em proprietário, aplicação e somente leitura basta até o segundo serviço ou a primeira coluna sensível. Faça o esquema crescer quando “quem executou isso?” não puder ser respondido pelo log, ou quando a credencial de um workload puder causar mais dano do que o seu trabalho justifica.

Erros comuns

- **Tratar o papel de aplicação como proprietário.** É tentador porque as migrações então rodam na inicialização com uma etapa a menos. O

serviço passa a deter o poder de remover e conceder de novo. Use um migrator separado e um proprietário sem login.

- **Conceder para corresponder às consultas do ano passado.** Um papel copiado do login existente mais amplo mantém todo privilégio que já foi conveniente. Derive o papel do que o workload faz agora, observado sob o seu próprio login.
- **Adicionar um papel por pessoa.** Papéis são pacotes de privilégios, e logins são por pessoa. Um papel com o nome de um indivíduo transforma o esquema em uma lista de funcionários. Mantenha os papéis voltados a funções.
- **Usar um papel para vários serviços porque são “parecidos”.** Os serviços se afastam um do outro e o papel mantém a união do que cada um precisa. Um papel por serviço custa algumas linhas.
- **Deixar padrões para tabelas futuras.** Um default privilege economiza uma linha em cada migração e expõe a próxima tabela sensível a um papel amplo. Mantenha o padrão vazio e conceda de forma explícita.
- **Conceder o papel do teto a um login por uma semana corrida.** O teto então se torna um grant permanente com um nome simpático. Mantenha o teto sem vínculos: somente a verificação das requisições o lê, e o teste de drift sinaliza qualquer participação.
- **Esconder uma coluna sensível atrás de uma view e deixar a tabela legível.** A view não estreita nada para um papel que alcança a tabela base. Revogue na tabela e teste com um papel que não detenha nenhum privilégio de tabela.
- **Confiar em um esquema declarado que ninguém compara.** Papéis no controle de versão não impedem o `GRANT` das duas da manhã. Compare o declarado com o real em um agendamento e faça a comparação falhar de forma ruidosa quando não conseguir ler.
- **Usar o superusuário para trabalho de rotina.** Ele funciona sempre, e é por isso que se espalha. Quando uma tarefa de rotina precisar dele, adicione o privilégio que falta ao papel certo.

Pontos-chave

- Um login compartilhado destrói a evidência e o escopo ao mesmo tempo: o log não consegue dizer quem agiu, e o login precisa ter permissão para fazer tudo o que o seu chamador mais amplo faz.
- Mantenha separados identidade, login e papel: um principal tem uma identidade, conecta-se por meio de um login, e o login é membro de papéis; ninguém faz login como um papel.
- O `ledgerly_owner` é proprietário de tudo e não tem login, o `ledgerly_migrator` o assume apenas durante uma migração, e cada workload recebe o seu próprio papel `app_*` dimensionado para o seu trabalho.
- Comece todo papel sem privilégios, use grants em nível de coluna onde colunas sensíveis ficam ao lado de colunas comuns, mantenha os default privileges vazios e recorra a tabelas de secrets separadas somente onde o motor não consegue conceder em colunas.
- O `human_jit_write` é um teto que nunca é concedido a um login: cada requisição aprovada concede um subconjunto diretamente ao login de sessão, e nenhuma requisição pode incluir uma coluna Restricted.
- Papéis e grants pertencem ao controle de versão e a etapas revisadas do pipeline, e uma verificação de drift agendada, que falha de forma ruidosa quando não consegue ler o banco de dados, mantém juntos o estado declarado e o real.
- Cada camada custa algo a alguém: manutenção para os papéis, instruções negadas para os grants estreitos, carga de revisão para as listas de colunas e uma aprovação mais grosseira onde o motor não tem grants em nível de coluna.

Os papéis dizem o que um principal pode fazer depois de conectado, mas ainda não dizem nada sobre como um serviço prova quem é. O Capítulo 4 dá o passo seguinte: como as credenciais chegam aos workloads, por que credenciais dinâmicas e de curta duração mudam a forma do problema e o que substitui a senha no arquivo de configuração.

PARTE II

Acesso das aplicações

Capítulo 4 — Credenciais e secrets

Imagine uma tarde de sexta-feira na Ledgerly. Alguém da equipe de backend foi para outro grupo, e a pessoa de contato da segurança pergunta, com educação, se a senha de administrador compartilhada pode ser trocada. Alex, da equipe de plataforma, concorda que deveria. Em seguida, Alex começa a listar onde a senha mora: a configuração dos cinco serviços, as variáveis do pipeline, o passo de export do `reporting-job`, um gerenciador de senhas da equipe e as conexões salvas em vários clientes desktop.

Trocar a senha no banco de dados leva um segundo. Trocá-la em todos os lugares, no mesmo instante, significa reiniciar todos os serviços de forma coordenada em horário comercial, e Robin, que está de on-call neste fim de semana, faz a pergunta justa: o que acontece com as faturas em andamento? A troca é adiada para “uma janela tranquila” que nunca chega. A senha sobrevive à pessoa e, depois, à pessoa seguinte.

Este capítulo responde à pergunta que aquela sexta-feira levantou: como um serviço obtém uma credencial e como se troca uma sem causar uma indisponibilidade? Ele acrescenta o próximo passo do estado-alvo no exemplo corrente. As credenciais dos serviços passam a ser de curta duração ou dinâmicas e chegam aos workloads por identidade, e não por um arquivo compartilhado.

As credenciais pertencem à parte Identidade do modelo de cinco partes; são necessárias, mas não são a resposta. A tese do livro é que “O acesso a bancos de dados em produção não é um problema de distribuição de credenciais; é um problema de autorização, execução e responsabilização”. Tratar melhor as credenciais reduz quem pode obter um login. Isso não diz nada sobre o que um login pode fazer depois que entra, nem sobre o que ele fez, e essas são as tarefas dos capítulos sobre grants, execução e registros.

O problema e por que ele é difícil

Uma **credencial** é o que um principal apresenta para se autenticar; um **secret** é qualquer valor que concede acesso se for divulgado. Uma senha é as duas coisas. O Capítulo 3 deu a cada serviço o seu próprio papel, de modo que um login diz o que pode fazer. Este capítulo trata da outra metade: como o login prova quem é e por quanto tempo essa prova continua valendo.

Uma senha tem um ciclo de vida mesmo quando ninguém o projeta. Ela é criada, copiada para qualquer lugar que um serviço consiga ler, usada por meses e, no fim, trocada ou esquecida. Cada cópia é um lugar onde uma pessoa ou um processo pode lê-la, e a pergunta “quem pode ver esta senha?” tem tantas respostas quantas são as cópias. No estado inicial, a resposta é “praticamente toda a equipe de backend e qualquer coisa que consiga ler a configuração de um serviço”.

O remédio óbvio é uma senha melhor: mais longa, aleatória, guardada em um lugar mais organizado. Isso ajuda contra tentativas de adivinhação, e adivinhar nunca foi o principal risco. O risco é o número de lugares onde o valor está e o fato de que nenhum deles pode ser esvaziado sem mexer nos outros. Uma senha forte em cinco arquivos de configuração continua sendo cinco cópias.

A segunda dificuldade é a **rotação**, que significa substituir uma credencial por outra nova, em um calendário ou por suspeita. A rotação é difícil porque uma credencial é estado compartilhado entre dois sistemas: o banco de dados conhece o valor, e cada consumidor também. Mudar um dos lados primeiro quebra o outro. Uma equipe que não consegue trocar uma senha sem uma indisponibilidade vai, com razão, parar de trocá-la, e uma política de rotação que ninguém consegue executar acrescenta uma entrada no calendário e não remove nenhuma exposição.

O que as credenciais não conseguem fazer

Pegue um login do reporting job, válido por uma hora, vazado em um canal de chat. Ele é uma boa credencial por todas as medidas deste capítulo: é único, tem curta duração e é obtido por identidade. Durante aquela hora, ele ainda pode ler as três colunas de `customers` que o papel recebeu (`id`, `created_at` e `status`), em todas as linhas, porque é isso que

`app_reporting_job` tem permissão de ler. Durações curtas reduzem a janela; não reduzem o papel. Três coisas delimitam um vazamento, e nenhuma delas é a credencial em si: o **papel** que ela carrega (Capítulo 3), o **alcance de rede** que decide de onde ela pode ser usada (Capítulo 7) e o **log** que registra o que ela fez (Capítulo 14). Um login que só pode ser apresentado a partir da rede do pooler, com um papel que não enxerga colunas `Restricted`, é um incidente menor do que qualquer política de senhas conseguiria produzir.

O mesmo vale para o melhor design apresentado a seguir. Um broker, um lease e uma workload identity respondem a quem pode obter um login. Eles não decidem o que o login pode fazer e não impedem que um login permitido execute uma instrução danosa.

Esta é uma questão de engenharia, não de política. Uma regra que diz “rotação a cada 90 dias” não diz nada sobre como os cinco serviços aprendem o novo valor no mesmo instante. A resposta está em como as credenciais são emitidas, entregues e substituídas, e isso é um design que a equipe precisa construir.

A vida de uma credencial

Ajuda acompanhar uma credencial do nascimento à aposentadoria e perguntar, em cada parada, quem consegue lê-la e o que a revelaria. Os lugares habituais são estes.

Lugar	Quem consegue ler	Como costuma vazar	Lugar melhor
Arquivo de configuração no repositório	Todos com acesso ao repositório, para sempre, pelo histórico	Copiado para um fork, um chat, um ticket	Um secret manager, referenciado pelo nome
Variável de ambiente	O processo, seus filhos e qualquer pessoa que consiga inspecionar o processo ou seus crash dumps	Impresso por um endpoint de debug ou por um relatório de erro	Um arquivo ou memória entregue na inicialização, com permissões de arquivo restritas
Imagem de contêiner ou artefato de build	Todos que conseguem baixar a imagem	Embutido no momento do build, de modo que toda cópia o carregue	Nada na imagem; obtido na inicialização

Lugar	Quem consegue ler	Como costuma vazar	Lugar melhor
Variável de CI/CD e log de job	Administradores do pipeline; às vezes todos os jobs	Exibido por um script, impresso por um comando que falha	Valor de curta duração obtido pela identidade do próprio job
Gerenciador de senhas da equipe	Todos no cofre compartilhado	Compartilhado por hábito; sem registro de quem leu	Acesso individual com registro, para os poucos casos que restarem
Conexão salva em um cliente desktop	Quem usar aquele laptop	Laptop perdido ou com backup	Nenhuma credencial salva; credenciais de sessão vindas de um gateway (Capítulo 7)

Duas coisas se destacam na tabela. Primeiro, os vazamentos quase nunca são engenhosos. São cópias feitas por conveniência: um valor colado em um ticket para que um colega reproduza um problema, uma variável exibida durante a depuração de um deploy. Segundo, a coluna “Lugar melhor” traz, em sua maior parte, a mesma resposta: o valor vive em um único lugar gerenciado, e os consumidores o buscam quando precisam.

Aviso: Um secret que apareceu em uma camada de imagem, no histórico de um repositório ou em um log de job precisa ser tratado como divulgado, mesmo que ninguém o tenha lido. Apagar a linha não apaga as cópias. O remédio é substituir a credencial, e é por isso que uma equipe precisa de um caminho de rotação que funcione antes do dia em que ele for necessário.

Um secret tem mais uma propriedade que vale nomear: por quanto tempo ele continua válido. Digamos que uma senha seja válida por três anos: ela carrega três anos de exposição de cada cópia já feita. Uma cópia antiga de uma credencial que dura uma hora não vale nada. As durações são ilustrativas. É por isso que a duração importa mais do que a força.

Secret managers e como um serviço lê um secret

Um **secret manager** é um serviço que guarda secrets, controla quem pode ler cada um e registra toda leitura. O HashiCorp Vault e os cofres de secrets dos grandes provedores de nuvem são exemplos. O padrão importa

mais do que o produto: um lugar guarda o valor, o acesso é decidido por principal e uma leitura deixa um registro.

Mover a senha para um secret manager não reduz, por si só, o número de leitores. Muda quem eles são e dá um nome a cada leitura: uma identidade de serviço específica lê um path específico, e o secret manager diz quando. É um ganho real para investigação e revogação, e um ganho modesto para a exposição.

O primeiro secret

Um serviço precisa de algo para provar quem é ao secret manager antes de poder ler qualquer coisa. Se esse algo for outra senha guardada, o problema só subiu um nível e nada foi resolvido. Essa é a questão do bootstrap, às vezes chamada de problema do primeiro secret.

A resposta é a **workload identity**: uma identidade que a plataforma dá a um workload em execução, como uma instância de serviço ou um job, para que ele possa se autenticar sem uma senha guardada. A plataforma que inicia o workload responde por ele, e o secret manager confia nessa declaração. O serviço nunca guarda um valor de longa duração só seu para esse passo. Recomendo tratar a workload identity como aquilo que torna possível o restante do design, porque ela encerra a regressão.

Ler na inicialização e ler de novo

Um serviço tem dois momentos razoáveis para ler a credencial do banco de dados: uma vez, na inicialização, ou sempre que abre uma conexão. Ler uma vez é simples e funciona com credenciais estáticas que raramente mudam. Falha quando a credencial muda, porque o processo em execução mantém o valor antigo até reiniciar. Ler no momento da conexão, por meio de um cliente que pergunta de novo ao secret manager depois de uma falha de autenticação, lida com a rotação sem reiniciar.

```

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

```

O bloco mostra as duas falhas que importam. Uma falha de autenticação dispara uma nova leitura, de modo que uma rotação não vira uma indisponibilidade. Um secret manager inalcançável recusa a inicialização, porque recorrer a uma cópia reserva recria o arquivo que este design removeu. Essa recusa tem um preço, e a seção Trade-offs e custos volta a ele.

Nota: O próprio secret manager é uma porta. Quem pode ler o path `db/billing-api`, quem pode alterá-lo e se as leituras são registradas em log são as mesmas perguntas feitas para o banco de dados. Conceder a todos os serviços acesso de leitura a todos os paths reconstruiria a senha compartilhada com um passo a mais.

Rotação sem indisponibilidade

A cura para “mude um lado primeiro e o outro quebra” é nunca ter um momento em que apenas um valor funcione. Esse é o **padrão de duas credenciais**, muitas vezes chamado de rotação A/B. Cada serviço tem dois logins, A e B, ambos membros do mesmo papel `app_*`. A qualquer momento, um está ativo e o outro está ocioso.

Para rotacionar, você altera o login ocioso, publica o novo valor, avisa os consumidores para trocarem, espera até que o login antigo não esteja mais em uso e só então o invalida. Em nenhum momento um consumidor segui-

ra um valor que o banco de dados deixou de aceitar. Em pseudocódigo, com os pontos de falha 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" })
```

Três detalhes sustentam o design. Primeiro, o banco de dados e o secret manager são sistemas separados, sem transação compartilhada, de modo que a ordem dos passos é o que limita uma falha: o valor que está em uso nunca é alterado antes que seu substituto possa ser lido. Uma falha em qualquer passo deixa a credencial anterior funcionando. Segundo, a verificação de esvaziamento. A rotação só termina quando nada usa o login antigo, e um serviço que nunca relê o seu secret é descoberto naquele momento, e não em uma indisponibilidade três semanas depois. Terceiro, o

último passo sobrescreve o login antigo com um valor que ninguém guarda, mas só roda depois do esvaziamento. Enquanto qualquer sessão ainda estiver com o login antigo, o valor antigo funciona.

O custo é que cada serviço passa a ter dois logins e um job que conhece os dois, e o consumidor precisa reler o secret depois de uma falha. Um serviço que lê a credencial uma única vez na inicialização precisa de um rolling restart para rotacionar, e esse restart é o ponto em que uma rotação A/B volta a ser um evento de calendário. Rotacionar a senha de administrador compartilhada no estado inicial é difícil por esse motivo: não existe um segundo login, e nenhum consumidor consegue reler coisa alguma.

Recomendo rotacionar em um calendário que a equipe consiga sustentar, e não no mais curto que ela consiga escrever. Uma suspeita de vazamento é outro caminho, descrito em “Quando uma credencial vaza”.

Credenciais dinâmicas e de curta duração

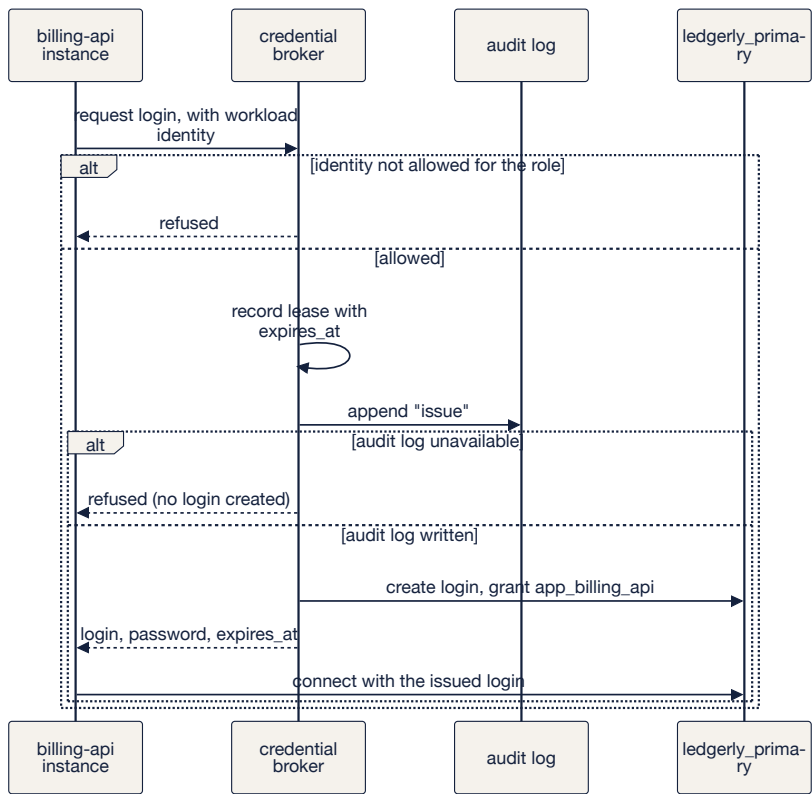
Uma credencial estática rotacionada ainda tem uma vida longa entre as rotações. O passo seguinte é tornar a duração curta o bastante para que a rotação deixe de ser um evento. Uma **credencial de curta duração** expira sozinha em um tempo curto e definido. Uma **credencial dinâmica** é uma credencial de curta duração gerada sob demanda para um único principal, de modo que duas requisições nunca recebem o mesmo valor. Toda credencial dinâmica é de curta duração; nem toda credencial de curta duração é dinâmica.

O mecanismo usual é um **credential broker**: um serviço que, sob demanda, cria um login no banco de dados, o inclui como membro do papel `app_*` correto e entrega a credencial do login ao workload solicitante junto com um **lease**. Um lease é um registro que diz quem detém a credencial, para qual papel e até quando. Quando o lease expira, ou alguém o revoga, o broker remove o login.

Na Ledgerly, o broker se conecta com o seu próprio login, membro de `ledgerly_access_admin`, o papel que cria e remove logins e gerencia as participações em papéis. Ele não possui tabelas, não é o proprietário e ninguém o usa para trabalho com dados. Esse é um poder mais estreito do que a propriedade, e ainda assim é amplo: quem detém o papel pode conceder qualquer coisa dentro dos papéis que ele administra, inclusive os pa-

péis de aplicação e os humanos. O broker precisa de apenas uma pequena parte desse poder. Dar-lhe menos significaria dividir o papel, o que tem custos próprios; aqui o broker usa o papel como ele é. O papel é um só, usado por três principais, cada um por meio do seu próprio login: o passo do pipeline, este broker e o gateway. O Capítulo 3 diz o que limita esse poder. Para o broker, a proteção é a que se afirma aqui: a credencial dele é guardada como a do proprietário, só a sua própria workload identity consegue obtê-la, e toda ação que ele executa vai para o log de auditoria (Capítulo 14).

A sequência a seguir mostra uma instância de `billing-api` pedindo um login. Observe os dois pontos em que a requisição é recusada: antes de qualquer coisa existir e quando o log de auditoria não pode ser alcançado.



Em pseudocódigo, a ordem das operações é o que mantém a sequência correta. Nomes de função como `createLogin` são ilustrativos, como no restante do livro.


```

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 }

```

O lease é escrito antes de o login existir, e a entrada de auditoria é escrita antes de o login ser criado. Se o banco de dados falhar depois que o lease for registrado, o sweeper encontra um lease sem login e o trata como já revogado. Se o log de auditoria estiver fora do ar, nenhuma credencial é emitida. É uma escolha deliberada, e significa que um log de auditoria inalcançável impede a inicialização de novas instâncias, um custo que o Capítulo 14 discute pelo outro lado.

Expiração e revogação

Uma expiração apenas impede que novas conexões se autenticuem. Uma sessão que já está aberta continua rodando depois que a senha expirou, de modo que a expiração tem duas partes: o login é invalidado e as sessões abertas nele são encerradas. Essa é também a diferença entre rotação e **revogação**, que faz uma credencial ou um grant parar de funcionar agora. A rotação substitui; a revogação encerra.

```

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")

```

A renovação segue a mesma disciplina. Uma instância em execução pede ao broker que estenda o lease antes de ele terminar, e o broker concorda apenas enquanto o lease estiver vivo e a duração total estiver abaixo de um teto. Sem um teto, a renovação transforma uma credencial de curta duração em uma de longa duração que por acaso é reemitida. A duração em si é uma decisão de julgamento. Não sei qual é o número certo para o seu sistema, e ele depende de três coisas: quanto tempo uma instância roda entre reinicializações, quantas conexões cada instância mantém e quanto tempo uma renovação leva quando o broker está sob carga. Apenas como ilustração, pense em um lease de uma hora renovado a cada trinta minutos.

O ganho em relação à rotação A/B é que uma credencial vazada expira sozinha. O login nomeia a instância, já que foi criado para ela, de modo que o log do banco de dados responde “quem executou essa instrução?” até o nível do workload. Isso entrega o que o Capítulo 3 prometeu: um login que corresponde a uma única instância de workload.

Workload identity e autenticação de banco de dados baseada em IAM

Existe um jeito de eliminar a senha guardada por completo. Algumas plataformas permitem que o banco de dados confie no próprio sistema de identidade da plataforma: o workload pede à plataforma um token de curta duração, e o banco de dados aceita esse token no lugar de uma senha. Os provedores de nuvem oferecem isso com nomes como IAM database authentication, e alguns bancos de dados conseguem verificar tokens de um provedor de identidade externo.

O atrativo é que não há secret para guardar, buscar ou rotacionar. O token é emitido para o workload quando necessário, tem curta duração (tipicamente minutos) e está vinculado à identidade que a plataforma já conhece. A revogação é tarefa da plataforma: remover o direito da identidade de pedir tokens impede novos tokens, e os tokens já emitidos podem ser aceitos até expirar. O login no banco de dados continua existindo, continua sendo membro do seu papel `app_*` e continua aparecendo no log.

Os custos são específicos:

- **Acoplamento à plataforma.** O design funciona onde a plataforma o oferece e em nenhum outro lugar, de modo que uma equipe com workloads em duas plataformas constrói o padrão duas vezes.
- **Mapeamento.** Alguém precisa dizer qual identidade da plataforma corresponde a qual login e a qual papel. Esse mapeamento é um grant e merece a mesma revisão que qualquer outro (Capítulo 14).
- **Conexões abertas.** Nos motores comuns, o token é verificado quando uma conexão é aberta. Como em qualquer expiração, uma conexão aberta sobrevive ao seu token (veja “Expiração e revogação”), de modo que ainda é preciso um limite para a vida útil da conexão (Capítulo 5).

Qual das três abordagens usar depende menos da moda do que do que a plataforma oferece e do que a equipe consegue operar.

Abordagem	Secret guardado?	Duração	Requer	Principal custo
Credencial estática em um secret manager, rotação A/B	Sim, um par por serviço	Até a próxima rotação	Secret manager, job de rotação	Os consumidores precisam re-ler; o job de rotação precisa continuar rodando
Credenciais dinâmicas de um broker	Apenas o do próprio broker	De minutos a horas, por instância	Broker, <code>ledgerly_access_admin</code> , lease sweeper	O broker é um componente de alto valor e sempre disponível
Tokens de workload identity	Nenhum para o banco de dados	Curta duração, tipicamente minutos	Suporte da plataforma e do banco de dados	Acoplamento à plataforma; tokens verificados apenas na conexão

Nenhuma delas remove o problema das credenciais; cada uma o desloca. O login do próprio broker é uma credencial estática, guardada no secret manager, obtida pela workload identity do broker e rotacionada em A/B. O design com tokens depende de o sistema de identidade da plataforma estar correto. O que muda é quantos secrets existem, por quanto tempo cada um é útil e quem pode lê-los.

TLS e verificação de certificados

Tudo o que foi dito até aqui diz respeito a onde uma credencial é guardada e quanto tempo ela dura. Antes de chegar ao banco de dados, ela atravessa uma rede, e uma conexão que não é criptografada carrega a senha, e os dados, de uma forma que qualquer coisa no caminho consegue ler. A criptografia em trânsito resolve isso e é a linha de base. Ela precisa valer em todos os trechos. Uma requisição costuma passar por um pooler ou por um gateway antes de chegar ao banco de dados, e cada um deles é uma conexão separada, com a sua própria criptografia e a sua própria verificação de certificado. Criptografar o trecho do cliente até o pooler não diz nada sobre o trecho do pooler até o banco de dados.

Há uma segunda tarefa, mais fácil de pular: saber que o servidor é o certo. Uma conexão pode ser criptografada até um destino que não é o banco de dados. A **verificação de certificado** é o cliente conferir que o certificado do servidor foi emitido por uma autoridade em que ele confia e que o certificado nomeia o host que ele pretendia alcançar. Sem ela, a criptografia protege contra quem lê o tráfego e não faz nada contra quem atende a conexão.

A configuração que importa é o modo de verificação do cliente. A maioria dos clientes de banco de dados oferece um modo que criptografa, mas não verifica, e ele é popular porque funciona com qualquer certificado, inclusive expirados e autoassinados. Considero esse modo metade do trabalho: ele esconde o tráfego de um observador passivo e dá a falsa sensação de que o servidor é o certo. O remédio é distribuir a autoridade certificadora para cada serviço e verificar o nome.

Isso traz custos próprios. Certificados expiram, e um certificado de servidor expirado é uma indisponibilidade com data exata, de modo que a expiração precisa de monitoramento e de um procedimento de renovação ensaiado. Distribuir o certificado da autoridade para todos os consumidores dá trabalho, e ele precisa ser atualizado quando a autoridade muda. Certificados de cliente, em que o certificado é a credencial, só funcionam com durações curtas se algo os emitir automaticamente.

Quando uma credencial vaza

Mesmo com todos os controles acima, uma credencial pode vaziar, e o design deve dizer de antemão o que acontece em seguida. O primeiro passo é separar as duas perguntas que o pânico mistura: como fazemos a credencial parar de funcionar e o que ela fez enquanto funcionava?

Para uma credencial dinâmica, a primeira resposta é uma revogação: encerrar o lease, fechar as sessões, e o login desaparece no tempo que o broker leva para agir. Para uma credencial estática em uma configuração A/B, uma suspeita de vazamento não é a rotação planejada. O job planejado espera o login antigo esvaziar, e um valor vazado continua válido enquanto alguém mantiver uma sessão nele. Em um vazamento, corte primeiro o login vazado: remova os papéis dele, feche as sessões e mude a senha sem esperar o esvaziamento; depois, publique um novo valor no login ocioso e

troque o ponteiro ativo. As instâncias que ainda seguram o login antigo perdem as conexões e falham até relerem o secret. Isso é um custo de disponibilidade, e o pagaria: uma credencial vazada que continua funcionando é o problema maior. Um serviço que não consegue reler o seu secret precisa de um restart, e a equipe deve saber quais são esses serviços antes do dia em que isso importar. Para uma senha de administrador compartilhada no estado inicial, a primeira resposta é a indisponibilidade com que este capítulo começou, o que é o argumento mais forte para fazer o trabalho antes de um incidente.

A segunda pergunta é respondida pelo log. Um login que corresponde a uma única instância de workload permite ver as instruções que ele executou, as tabelas que tocou e as horas em que esteve ativo. Isso restringe a revisão ao raio de impacto de um papel, e não ao banco de dados inteiro. O procedimento passo a passo, a ordem em que agir e como ensaiá-lo pertencem ao Capítulo 15, que trata do acesso durante incidentes. Aqui, o ponto é que uma equipe que construiu a revogação como um passo de rotina pode usá-la em um dia ruim sem precisar inventá-la.

Os regimes de proteção de dados e de cartões de pagamento tendem a perguntar como o acesso é concedido, quanto tempo dura e como é revogado. Os leases, os passos de revogação e as entradas de auditoria descritos aqui são os registros que respondem a essas perguntas.

O plano de migração da Ledgerly

A passagem da senha de administrador compartilhada para credenciais de curta duração é uma sequência de passos pequenos e reversíveis. Recomendando ordená-los por quanto risco cada passo remove e por quão pouco ele pode quebrar.

1. Faça o inventário de todas as cópias da senha de administrador.

Serviços, variáveis do pipeline, o passo de export, o gerenciador de senhas, conexões salvas em ferramentas de cliente. A lista é o plano de trabalho. O inventário do Capítulo 1 é o ponto de partida.

2. Crie um login por serviço, no seu papel `app_*`. Use os papéis do Capítulo 3. Nesta etapa, as credenciais são estáticas, mas cada uma é separada, guardada no secret manager e lida por identidade.

3. **Tire os serviços do login de administrador, um de cada vez.** Comece pelo `reporting-job`, cujo raio de impacto é o menor, depois `invoice-worker`, `admin-console` e `billing-api`. Faça o `payments-service` por último, porque é o único serviço que alcança `payment_tokens` e `bank_accounts`, e um erro ali custa mais. Cada mudança é um deploy que pode ser revertido.
4. **Dê às migrações e ao passo de export os seus próprios logins.** O pipeline usa um login de curta duração em `ledgerly_migrator` (Capítulo 11); o papel em si mantém privilégios permanentes de proprietário, e apenas a credencial é de curta duração. O passo de export usa `app_vendor_export` (Capítulo 13). Nenhum dos dois usa o login de administrador depois deste passo.
5. **Passe cada serviço para a rotação A/B.** Faça cada serviço reler o seu secret depois de uma falha de autenticação, execute a primeira rotação em um período tranquilo e observe a verificação de esvaziamento. Um serviço que não esvazia é um achado, e o achado é o objetivo deste passo.
6. **Introduza o broker para os serviços que renovam com facilidade.** Comece por um serviço que reinicia com frequência e mantém poucas conexões. Acrescente os outros conforme a equipe aprender como os leases se comportam. Mantenha a A/B estática como fallback.
7. **Aposente o login de administrador compartilhado.** Remova-o de toda configuração e do gerenciador de senhas. Depois que a propriedade tiver passado para `ledgerly_owner` (Capítulo 3), retire os privilégios dele e remova-o, ou desabilite-o. Se o administrador for o superusuário do motor, como costuma ser o usuário master de um serviço gerenciado, ele se torna o que o Capítulo 3 chama de instrumento selado. É uma credencial que ninguém usa no trabalho diário. Ela fica guardada onde a equipe de plataforma consegue recuperá-la, e cada recuperação é registrada. Troque a credencial dele uma vez, depois que todo o inventário estiver limpo.

Uma equipe poderia parar depois do passo cinco e já estar muito melhor: cada serviço tem a sua própria credencial, a lê por meio de uma identidade e a rotaciona sem indisponibilidade. O broker é um acréscimo para equipes que conseguem operá-lo.

Trade-offs e custos

Cada movimento deste capítulo desloca custo de um incidente para uma rotina. Veja onde a rotina fica cara.

Dependência do secret manager. Quando as credenciais passam a vir de um secret manager, um secret manager inalcançável impede a inicialização de novas instâncias e, com leases curtos, acaba impedindo as instâncias em execução de renovar. Isso o torna parte da disponibilidade do caminho até o banco de dados. Opere-o com o cuidado que você dá ao banco de dados: mais de uma instância, uma recuperação testada e um rodízio de on-call que saiba que ele existe. Manter uma credencial em cache pela duração de um lease suaviza uma indisponibilidade breve e aumenta o tempo em que uma credencial revogada ainda pode ser usada.

Connection storms. Se todas as instâncias de um serviço renovam no mesmo minuto, ou reiniciam juntas depois de um deploy, o banco de dados vê uma rajada de novas conexões e autenticações. A autenticação não é gratuita, e uma rajada provocada pelo próprio sistema pode deixar o banco de dados lento para todos os outros papéis, uma connection storm sem nenhuma causa externa. Espalhe as renovações com jitter, mantenha estável o pool de conexões entre as renovações e deixe o Capítulo 5 definir quantas conexões um papel pode manter.

Depuração mais difícil. Um login que existe por uma hora já desapareceu quando alguém lê o log na manhã seguinte, e um nome como `billing-api-4f2a` não significa nada sem o registro do lease. Guarde os leases e a identidade correspondente pelo mesmo tempo que o log de auditoria é guardado e dê a quem estiver de on-call uma consulta que leve do login à instância. Se essa consulta não existir, a credencial de curta duração torna a investigação mais lenta, e não mais rápida.

Um broker de alto valor. O login do broker pode criar logins em todos os papéis de aplicação, de modo que vale o esforço de um atacante. Mantenha o código dele pequeno e revisado e dê-lhe a atenção que você dá àquilo que ele protege; um broker que recebe menos é um risco em si.

Quando não usar isto. Uma equipe com um único serviço, um punhado de engenheiros e um cofre de secrets gerenciado pode ser bem atendida por uma credencial estática separada, guardada no secret manager e rotacionada em A/B manualmente duas vezes por ano. Um broker é mais um

serviço para operar e mais um sistema para a equipe aprender; abaixo de um certo tamanho, as partes móveis adicionais vão causar mais indisponibilidades do que evitar. Se você não consegue dizer quem receberia o page quando o broker estiver fora do ar, esperaria. Os tokens de workload identity são o passo mais barato quando a plataforma os oferece, e ainda exigem que o mapeamento seja revisado.

Erros comuns

- **Tratar o secret manager como a resposta completa.** É tentador porque a mudança é visível e rápida. A senha compartilhada continua compartilhada se todos os serviços leem o mesmo path. Dê a cada serviço o seu próprio path e o seu próprio login e revise quem pode ler cada um.
- **Uma rotação que ninguém consome.** Um job rotaciona o valor, mas os serviços o leem na inicialização e nunca mais, de modo que o novo valor fica sem ser lido até o antigo ser aposentado e algo falhar. Teste se os consumidores absorvem uma rotação e faça a verificação de esvaziamento parte do job.
- **Recorrer a um valor embutido.** Uma cópia “última conhecida como boa” na imagem torna a inicialização confiável e traz de volta o arquivo que o design removeu. Recuse a inicialização, envie um alerta e conserte a dependência.
- **Criptografar sem verificar.** Ativar o TLS cumpre um checklist, e o modo que só criptografa faz a conexão funcionar. Verifique o certificado e o nome do host e monitore a data de expiração.
- **Durações sem teto.** Uma renovação sem limite reconstrói uma credencial de longa duração. Limite a duração total de um lease e faça a instância se reautenticar com a sua identidade.
- **Tratar o papel do broker como pequeno.** `ledgerly_access_admin` é mais amplo do que a tarefa do broker, de modo que um broker que for tomado pode chegar tão longe quanto os papéis que ele administra. Restrinja quais principais podem chamá-lo e quais papéis `app_*` cada um pode solicitar, e revise as entradas de auditoria dele. Trate o papel amplo como um custo que você aceitou, não como um limite.

Pontos-chave

- Cada cópia de uma credencial é um leitor, de modo que a mudança mais útil é reduzir o número de cópias e dar um nome a cada leitura que restar.
- A rotação é difícil porque uma credencial é estado compartilhado; dois logins por serviço tornam possível alterar um enquanto o outro continua funcionando.
- Uma rotação que os consumidores não absorvem, ou que ninguém executa, é uma entrada de calendário e não um controle, de modo que a verificação de esvaziamento e um alerta para um job de rotação que parou de rodar fazem parte do design.
- As credenciais dinâmicas dão a cada instância de workload o seu próprio login, com um fim embutido, ao preço de um broker, de um registro de lease e de uma dependência do secret manager.
- A workload identity remove a senha guardada onde a plataforma a oferece e desloca a confiança para a plataforma e para o mapeamento entre identidade e login.
- Verificar o certificado e o nome do servidor é o que faz o TLS proteger a credencial; a criptografia sem verificação esconde o tráfego e deixa a identidade do servidor sem conferência.
- Quando uma credencial vaza, a velocidade da revogação e a clareza do log decidem o quanto o evento fica contido, e isso depende de trabalho feito antes do vazamento.

As credenciais decidem quem entra; não dizem nada sobre quantas conexões elas abrem nem por quanto tempo uma instrução pode segurar o primary. O Capítulo 5 transforma a própria conexão em uma política: limites por papel, pooling, timeouts e os rótulos que facilitam encontrar uma consulta descontrolada.

Capítulo 5 — Conexões como política

Considere a Ledgerly em uma terça-feira tranquila, à noite. Um colega aumentou o paralelismo da agregação noturna no `reporting-job` para que ela terminasse antes da manhã, uma mudança que parecia inofensiva na revisão. À 01:40 o job começa, abre uma conexão para cada uma de suas threads de trabalho e passa a ler `payments` e `invoices` do `primary`, usando o login `admin` compartilhado. Em poucos minutos, o `billing-api` começa a devolver erros a clientes de um fuso horário em que já é manhã. Robin, de on-call, recebe um page por uma indisponibilidade que afeta clientes.

Robin abre o banco de dados e descobre que nada está quebrado. O `primary` está no ar, a atividade de disco está normal e nenhuma instrução fez nada de errado. O servidor tem um número fixo de slots de conexão, o relatório ficou com a maior parte deles e a API espera na porta. Como todos os chamadores usam o mesmo login, o banco de dados não consegue dizer que o relatório é a causa. Só consegue dizer que os slots estão cheios.

Este capítulo responde a uma pergunta que fica entre a identidade e os dados: como fazemos da própria conexão um lugar onde a política é aplicada? No modelo de cinco partes do Capítulo 1, ele atua sobre duas partes. A conexão é o Caminho, porque o pooler decide como os serviços chegam e quantos deles podem chegar. Ela também é o primeiro limite da Execução, porque limites e timeouts dizem quanto um chamador, e uma instrução, pode tirar de todos os outros. O passo do exemplo corrente é que as conexões passam a ser moldadas por um pooler e por limites e timeouts por papel.

O problema e por que ele é difícil

Uma conexão é um recurso antes de ser qualquer outra coisa. Na maioria dos motores, cada uma custa ao servidor memória e uma fatia do seu escalonamento, e o servidor permite apenas um número fixo delas. O limite é uma propriedade do servidor, não do chamador, de modo que o primeiro chamador a pedir muitas recebe muitas. Nada na configuração padrão diz que o relatório noturno deveria ter menos slots do que a API que atende os clientes.

Uma senha não toca em nada disso. Ela decide quem entra; não diz nada sobre quantos slots a pessoa ocupa nem por quanto tempo. É por isso que um livro que trata de mais do que distribuir credenciais precisa de um capítulo sobre conexões.

Três hábitos pioram o problema. Primeiro, o número de conexões é um produto, não uma soma: as instâncias de um serviço multiplicadas pelo tamanho do pool de cada instância, multiplicadas de novo sempre que um autoscaler acrescenta uma instância. Digamos que uma equipe configure um pool de vinte em um serviço que escala até dez instâncias. Ela pediu duzentas conexões e provavelmente não decidiu isso de propósito. Segundo, uma conexão é mantida enquanto a sua instrução roda, de modo que uma consulta lenta ocupa um slot o tempo todo, e slots são o que o próximo chamador precisa. Terceiro, um chamador que falha tende a insistir com mais força. Um cliente que sofre timeout e tenta de novo imediatamente transforma um banco de dados lento em uma multidão na porta.

A resposta óbvia é aumentar o limite do servidor. Isso ajuda por uma semana e depois torna o próximo incidente maior. Mais conexões significam mais trabalho concorrente no mesmo primary, e o banco de dados não fica mais rápido em executar cem instruções ao mesmo tempo só porque cem são permitidas.

Os controles ficam em vários lugares que ninguém administra em conjunto: a definição do papel, o pool da aplicação, um possível pooler e as configurações que decidem por quanto tempo uma instrução pode rodar. A disponibilidade é uma das propriedades em jogo no modelo de ameaças do Capítulo 2, e os acidentes a dominam. Um login compartilhado transforma todo acidente em acidente de todos.

No estado inicial, portanto, os cinco serviços e todo engenheiro conectado a uma VPN tiram de um único conjunto de slots do servidor, sob um único nome. Não há timeouts além do que cada biblioteca de cliente define por padrão, e não há registro de qual chamador segura qual conexão. As seções a seguir substituem isso por um orçamento por papel, um pooler que o aplica, timeouts que fixam um limite superior para o tempo que qualquer coisa pode segurar um slot e um rótulo em cada conexão que diz a quem ela pertence.

Um orçamento de conexões por papel

O primeiro passo é o mais barato. O Capítulo 3 deu a cada serviço o seu próprio papel de aplicação, o que significa que um limite finalmente pode ser associado a um nome. Um limite por papel diz: este principal pode manter no máximo este número de conexões, não importa quantas instâncias execute nem com que insistência peça. Quando ele atinge o limite, novas conexões desse papel são recusadas, e todos os outros papéis continuam funcionando.

Essa recusa é a política. Ela desloca a falha de “o banco de dados está cheio para todos” para “este chamador ficou sem slots”, e o segundo é um page muito melhor para Robin receber. O raio de impacto de um chamador ganancioso passa a ser o próprio chamador, e a recusa nomeia o chamador, que é a metade de responsabilização do mesmo passo.

Os números pedem raciocínio, não uma fórmula. Trabalharia a partir do limite do servidor para baixo. Primeiro, subtraia uma reserva que mais nada pode usar, porque as pessoas e as ferramentas que consertam o sistema também precisam se conectar. A reserva cobre as sessões da plataforma e os logins dos três principais que usam `ledgerly_access_admin` (o passo do pipeline que aplica as definições de papéis, o credential broker e o access gateway). Depois, divida o restante entre os papéis de aplicação conforme o que cada um precisa no seu pico, com alguma folga. A tabela a seguir é uma ilustração para o primary. Os números são hipotéticos, não uma recomendação.

Principal	Papel	Limite de conexões (ilustrativo)	Por que este tamanho
billing-api	app_billing_api	80	Serviço mais ocupado; atende o aplicativo web e a API pública
invoice-worker	app_invoice_worker	30	Dirigido por fila; a concorrência é definida pelo número de workers
payments-service	app_payments_service	30	Estável; conversa com provedores externos entre as instruções
admin-console	app_admin_console	15	Ações restritas e auditadas, feitas por poucas pessoas
Migrações	ledgerly_migrator	3	Uma migração por vez, a partir do pipeline
Sessões humanas no primary	Logins de sessão dos tipos JIT e break-glass, criados pelo gateway	8	Só rodam aqui sessões com uma requisição aprovada (Capítulo 6); as leituras de rotina vão para ledgerly_replica_2. Os papéis human_jit_write e human_breakglass não têm limite, porque nenhum login é membro do primeiro e o segundo é concedido por sessão
reporting-job	app_reporting_job	10	Apenas na transição: ele lê o primary no estado inicial e durante a mudança, e a linha vai a zero no estado-alvo
Reserva	administração de acessos e plataforma	10	Apenas para reparo e para a emissão de logins
Não alocado	nenhum	14	Folga, para que um erro de arredondamento não seja uma indisponibilidade

Suponha que o servidor permita 200 conexões. As linhas alocadas somam 186, e as 14 restantes são deixadas deliberadamente sem atribuição. As sessões humanas de rotina não estão nesta tabela porque não rodam no primary. O orçamento delas é definido em ledgerly_replica_2, para onde vão as leituras de rotina; digamos, 20 slots ali. A linha do reporting-job existe apenas enquanto o relatório ainda lê o primary. No

estado-alvo, ele lê de `ledgerly_replica_2`, com o seu próprio orçamento naquele host, definido na camada de conexão, no pooler (o Capítulo 6 trata dessa mudança). As definições de papéis poderiam carregar os limites em uma notação como esta, que é ilustrativa e não pertence a nenhum 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: Os motores diferem quanto ao que um limite ou uma configuração pode ser associado. Alguns contam conexões por papel, alguns por login e alguns apenas por banco de dados ou por servidor. Em alguns motores, um limite ou um valor padrão escrito em um papel se aplica apenas às conexões que fazem login como o próprio papel e não é herdado pelos logins que são membros dele. O Capítulo 4 dá a cada instância de serviço o seu próprio login dinâmico, membro do papel `app_*`, de modo que, nesses motores, o limite no nível do papel e os timeouts no nível do papel descritos a seguir não se aplicariam, sem nenhum aviso.

Dois lugares podem carregá-los no lugar: o broker, que pode definir um limite por login e as mesmas configurações quando cria cada login, e o pooler, que pode carregar as configurações mas talvez não consiga somar os logins de um papel em um único total. A seção sobre o pooler, adiante, apresenta três maneiras de obter o total por papel quando ele não consegue.

Duas propriedades tornam a tabela útil. Os limites são verificados onde a conexão começa, de modo que o chamador fica sabendo da recusa na hora, e não depois da primeira instrução lenta. E a reserva fica fora de todos os papéis de aplicação, para que, na pior noite, a equipe de plataforma ainda consiga fazer login e consertar as coisas. Um orçamento que preen-

che o servidor inteiro tem o mesmo defeito de uma senha de administrador compartilhada: funciona até o momento em que alguém mais precisa dele.

As contas não precisam ser conservadoras. Se `billing-api` e `payments-service` raramente têm picos ao mesmo tempo, você pode deixar os limites de aplicação somarem mais do que o servidor permite e aceitar que dois picos simultâneos possam colidir. Só faria isso com a reserva mantida de lado e só depois de a equipe observar o uso real por algum tempo. O padrão é um orçamento que fecha a conta, porque um orçamento que não pode ser violado é mais fácil de raciocinar às três da manhã.

Pooling: onde as conexões são compartilhadas

Um pool mantém um conjunto de conexões abertas e as empresta aos chamadores, para que cada requisição não pague por uma nova conexão e o banco de dados veja um número limitado delas. Há dois lugares onde colocar um.

Quase todo serviço já tem um pool na sua biblioteca de banco de dados, e ele deve ser dimensionado pelo trabalho que uma instância consegue fazer ao mesmo tempo, não pelo tráfego que ela espera atender. Um pool pequeno que faz os chamadores esperarem um pouco é mais gentil com o banco de dados do que um pool grande, e deve desistir depois de uma espera curta, com um erro que o serviço possa transformar em uma resposta clara. Comece pequeno, observe quanto tempo os chamadores esperam e aumente o tamanho com base em evidências.

O pooler na frente do banco de dados

Um **pooler** é um proxy de banco de dados cuja principal tarefa é aceitar muitas conexões de clientes e multiplexá-las sobre um número menor de conexões com o servidor. PgBouncer e ProxySQL são exemplos comuns. O serviço se conecta ao pooler como se ele fosse o banco de dados, e o pooler decide quando uma conexão real é emprestada.

A pergunta que o livro continua fazendo vale aqui também: que nome o banco de dados vê? O Capítulo 4 deu a cada instância de serviço o seu próprio login dinâmico, criado pelo credential broker e membro do papel `app_*` do serviço. O pooler precisa se encaixar nesse modelo, e o construi-

ria neste formato. A instância apresenta ao pooler o seu próprio login e a sua própria credencial. O pooler os verifica no banco de dados e mantém as conexões com o servidor sob esse mesmo login. Ele não tem um repositório de senhas próprio e não tem um login próprio que todos os clientes compartilhem.

Uma credencial rotacionada ou expirada chega ao pooler pelo banco de dados, não pelo pooler. Quando um lease termina ou é revogado, o broker remove o login (Capítulo 4), de modo que a próxima tentativa do pooler de se autenticar ou de abrir uma conexão com o servidor sob esse login é recusada, e ele fecha as conexões ociosas com o servidor que mantém para ele.

Como um limite por papel continua valendo não tem resposta universal, e não suporia que o seu pooler consegue dar uma. Muitos poolers mantêm os limites por par de login e banco de dados, e alguns acrescentam um limite por login ou por banco de dados. Não conheço nenhuma capacidade geral que some as conexões de muitos logins diferentes em um único teto no nível do papel, então leia a documentação do seu pooler em vez de aceitar a palavra deste capítulo. Se o seu não consegue, o total por papel precisa vir de outro lugar. Há três lugares, e cada um tem um preço.

- **O pooler se conecta ao upstream com um login estável por papel.**

A credencial dinâmica da instância passa a autenticar o cliente apenas perante o pooler, e o limite por login do pooler é exatamente o limite do papel. O preço é o modelo de identidade do Capítulo 3:

- O log do banco de dados mostra um nome por papel, de modo que a atribuição a uma instância passa para o log do pooler e para o rótulo da aplicação descrito adiante.
- O pooler precisa verificar por conta própria as credenciais dinâmicas, o que o torna um consumidor do estado dos leases do broker.
- Ele também guarda uma credencial de longa duração para cada papel, que precisa ser protegida e rotacionada em A/B, como descreve o Capítulo 4.
- Revogar o login de uma instância no broker deixa de fechar as conexões dela com o servidor; o pooler precisa encerrar as sessões desse cliente.

- **O banco de dados impõe um limite por papel, onde o motor oferece suporte a isso para logins membros.** É exato, fica onde nada consegue contorná-lo e os logins das instâncias e a atribuição a elas permanecem intactos. O preço é que o suporte varia, como diz a nota acima. Onde funciona, a recusa vem do banco de dados depois que a conexão já custou uma autenticação, e nada é multiplexado.
- **O broker limita quantos logins emite por papel.** Com um limite por login definido quando cada login é criado, o total do papel não pode exceder o número de logins vezes esse limite. O preço é que isso é um limite superior, e não uma contagem exata, e que o teto precisa corresponder ao teto de instâncias do autoscaler. Um scale-out além do teto vira um deploy que não consegue obter credenciais, o que é um problema de disponibilidade, e renovações que sobrepõem por pouco tempo um login antigo e um novo precisam de espaço abaixo do teto.

Tentaria primeiro o limite no lado do banco de dados, onde o motor o respeita para logins membros. Onde ele não respeita, o teto no broker preserva o modelo de identidade e custa um teto para manter. Só adotaria o login único no upstream por papel quando as contagens de conexões forcessem a multiplexação entre instâncias, e registraria por escrito o que o log deixa de dizer. Os três são melhores do que o design que mapeia todos os clientes de todos os papéis para um único login compartilhado. Ele recria a conta de administrador compartilhada uma camada abaixo: o log volta a mostrar um único nome, e um limite se aplica a todos de uma vez.

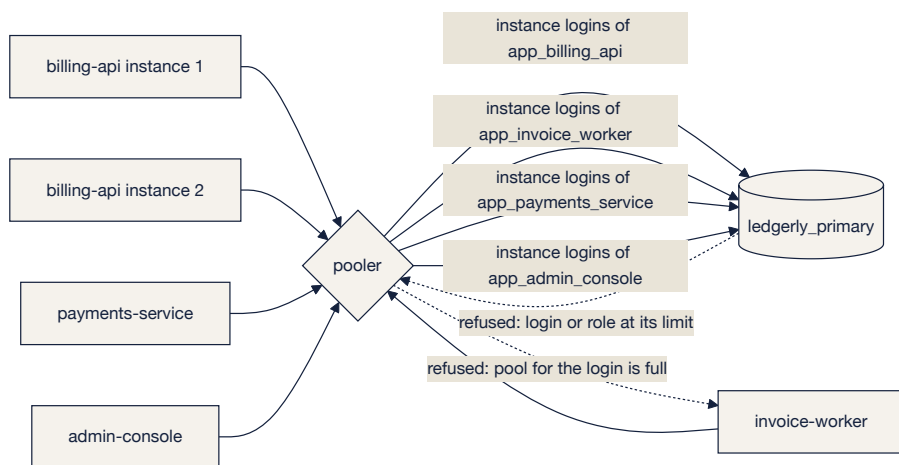
No formato pass-through recomendado, o pooler traz pouca multiplexação. No modo de transação, os clientes de uma instância compartilham as conexões dessa instância com o servidor, mas duas instâncias não compartilham entre si, de modo que o total de um papel é, no mínimo, o número de instâncias ativas vezes um pequeno mínimo. O que o pooler traz são três outras coisas.

- **Recusa central.** Um único lugar diz não, por login e por pool, da mesma forma para toda biblioteca de cliente.
- **Os mesmos limites para todos os serviços.** Os tamanhos de pool e as esperas de cada serviço deixam de ser a única linha de defesa.
- **Um chokepoint de rede.** Os serviços entram no banco de dados apenas pelo pooler. A política de rede, que diz quais hosts podem abrir uma conexão TCP e que o Capítulo 6 define e o Capítulo 7 escreve, ad-

mite os hosts do pooler e mais ninguém, de modo que uma credencial que vaza de um serviço é inútil a partir de qualquer lugar de onde o pooler não possa ser alcançado. O alcance delimita um vazamento, como diz o Capítulo 4, e o pooler é onde esse limite é fácil de enunciar.

O custo é um novo sistema no caminho, tratado nos trade-offs adiante. Para um único serviço pequeno com uma instância, o pool da aplicação e um limite por papel bastam, e o pooler não vale a pena até que o número de instâncias ou a necessidade de um único chokepoint digam o contrário.

O diagrama a seguir mostra o formato pass-through, que deixa os logins das instâncias visíveis para o banco de dados. Observe os dois lugares em que um chamador é recusado: no pooler, quando o pool de um login está cheio, e no banco de dados, quando um login ou, onde o motor oferece suporte, o papel está no seu limite.



Modo de sessão e modo de transação

Um pooler empresta uma conexão com o servidor por um período, e a duração desse período é o seu **modo**. No modo de sessão, o cliente mantém a mesma conexão com o servidor da conexão à desconexão; o pooler não economiza nada além do custo de conectar. No modo de transação, o cliente segura uma conexão com o servidor apenas pela duração de uma transação, e a transação seguinte pode rodar em outra. O modo de transação multiplexa muito mais clientes sobre poucas conexões, e é por isso que ele é atraente.

Ele vem com um custo fácil de perder, porque diz respeito à autorização e não ao desempenho. Tudo o que um cliente define na sua sessão vive em uma conexão com o servidor, e no modo de transação o cliente não é proprietário de nenhuma. As configurações feitas fora de uma transação podem desaparecer na instrução seguinte ou, pior, podem permanecer em uma conexão que o pooler empresta a outra pessoa. As coisas que dependem do estado da sessão incluem tabelas temporárias, prepared statements em muitas configurações, advisory locks, timeouts no nível da sessão e qualquer valor de contexto que o seu próprio design guarde na sessão.

Esse último item toca o livro diretamente. No Capítulo 3, uma regra de linha pode depender de um valor como `current_case_customer()`, e `app_admin_console` leva a identidade de um principal humano para o log de auditoria e para `audit_events`. Se qualquer um dos dois valores for definido uma vez por sessão e lido depois, então no modo de transação ele pode estar ausente ou, no pior caso, pertencer ao cliente anterior.

Aviso: No modo de transação, um valor de contexto definido no nível da sessão pode ser deixado em uma conexão com o servidor e lido pela transação do cliente seguinte. Uma regra de linha ou um registro de auditoria que o leia pode então agir pelo cliente errado ou pela pessoa errada. Defina esses valores dentro da transação, com escopo restrito a ela, e faça o código que os lê falhar quando o valor estiver ausente.

O padrão que se sustenta é definir o contexto dentro da transação e deixá-lo terminar com ela. O bloco a seguir mostra o formato, escrito para continuar correto quando o trabalho falha no meio.

```

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

```

O log do banco de dados continua mostrando o login da instância, mas no modo de transação os clientes de uma instância compartilham as conexões dela com o servidor, de modo que o log não consegue dizer qual das threads ou das requisições da instância segurou um slot. Para descobrir isso, é preciso a visão que o próprio pooler tem dos seus clientes, e ela precisa ser planejada antes do primeiro incidente. É por isso que as próximas seções, sobre timeouts e nomes, importam.

Os humanos não usam este pooler. Eles chegam ao banco de dados pelo access gateway, que cria um login para cada sessão (Capítulo 7). Os limites deles são o assunto da última parte da próxima seção, porque uma pessoa pode fazer algo que um serviço não pode: suspender o limite.

Timeouts como política

Um slot é emprestado enquanto uma instrução ou uma transação roda, e na maioria dos motores nada encerra nenhuma das duas por padrão. Os timeouts são a forma de a equipe de plataforma dizer por quanto tempo um papel pode segurar um recurso, e são a parte deste capítulo que delimita a Execução: uma instrução que roda por tempo demais é cancelada onde está rodando, e o slot é devolvido. Eles pertencem à definição do papel e à configuração do pool, e não à boa vontade de cada biblioteca de cliente, e é por isso que este capítulo os trata como política.

Três timeouts pertencem ao banco de dados e dois ao pool.

Timeout	Perten- ce a	O que ele interrompe	Quem sente
Statement	Banco de dados	Uma consulta que roda muito mais tempo do que o papel jamais deveria precisar	O chamador dessa instrução
Lock wait	Banco de dados	Uma instrução na fila atrás de um bloqueio, e todos que estão na fila atrás dela	O chamador; também qualquer chamador que espere pela mesma linha
Idle in transaction	Banco de dados	Uma transação aberta e depois abandonada, segurando bloqueios e slots	A aplicação que a abriu
Vida útil da conexão	Pool	Uma conexão que fica aberta por dias e mantém a credencial ou o estado antigos	O pool, que precisa se reconectar
Espera para obter conexão do pool	Pool	Um chamador esperando para sempre por uma conexão livre	O serviço que trata a requisição

Papéis diferentes precisam de valores diferentes, porque fazem trabalhos diferentes. Uma requisição no `billing-api` é uma pessoa esperando por uma página; uma instrução que precisa de trinta segundos ali é um bug. O `invoice-worker` pode legitimamente rodar uma instrução mais longa enquanto monta um lote. O `reporting-job` roda leituras longas por design, em uma réplica onde uma leitura longa não bloqueia os writers. Um valor padrão compartilhado estaria errado para todos eles, de modo que os valores são associados ao papel. O bloco a seguir usa uma notação ilustrativa, e os valores são 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;
```

A última linha reflete uma lição de migração que o Capítulo 11 trata: uma mudança de esquema pode rodar por muito tempo, mas deve esperar apenas brevemente por um bloqueio.

Duas regras sobre a ordem importam mais do que os valores individuais. O `statement timeout` do banco de dados deve ser menor do que o `deadline` de requisição da aplicação, porque, se a aplicação desistir primeiro, ela abandona uma instrução que ainda está rodando e o slot continua ocupado; e a espera para obter conexão do pool deve ser a menor de todas, para que os chamadores saibam do congestionamento cedo. Quando um `deadline` dispara primeiro, a conexão ainda pode estar carregando uma instrução em execução, então o serviço a fecha e não a empresta ao chamador seguinte.

A transação ociosa é a surpreendente. Um serviço que abre uma transação, chama um provedor de pagamentos externo e espera com a transação ainda aberta deixa a lentidão do provedor segurar bloqueios e um slot no banco de dados. A correção de longo prazo é manter as chamadas externas fora das transações, o que é uma mudança de código; o `timeout` mantém o custo limitado enquanto essa mudança está no backlog.

Uma conexão que vive por semanas mantém tudo com o que nasceu, de modo que um tempo de vida máximo faz o pool substituir suas conexões regularmente. Ele deve ser menor do que a duração da credencial, com alguma variação entre as conexões para que elas não se reconectem todas no mesmo minuto. O Capítulo 4 explica por que uma credencial expirada não fecha uma sessão que já está aberta; o tempo de vida é a forma de o pool fechar as suas próprias.

Valores padrão que uma sessão pode suspender

Um `timeout` definido como valor padrão de um papel é um valor padrão. Em muitos motores, uma sessão pode alterar as suas próprias configurações, inclusive o `statement timeout` e o modo somente leitura, e pode trocar de papel. Para um serviço isso é aceitável: os limites existem para acidentes, e o código não suspende o próprio `timeout` por acidente. Para uma pessoa com um cliente SQL, não é, porque a sessão que precisa do limite é a sessão que pode suspendê-lo.

Dois controles ficam onde a sessão não alcança. O gateway abre cada sessão humana com os seus limites aplicados e recusa a requisição de alterar as configurações de uma lista protegida; o Capítulo 9 trata de como o gateway faz isso. O custo é que os clientes que enviam essas configurações ao se conectar deixam de funcionar até que alguém as liste. As sessões humanas recebem os limites do gateway: ele define um limite de conexões por login conforme o tipo de sessão quando cria o login da sessão e conta quantas sessões uma pessoa pode manter. Nenhum dos dois é um limite de papel, já que `human_jit_write` é um teto do qual nenhum login é membro. Nenhum dos dois é deixado ao cliente. Para qualquer limite que não tenha nenhum dos dois, a tabela de controles deveria dizer que ele é um valor padrão e não uma garantia.

Dar nome a quem chama

Quando Robin olha para as conexões, a pergunta útil é “de quem é esta?”, e o login responde à primeira metade. Um papel diz que uma conexão pertence ao `billing-api`. Não diz qual instância, qual release ou qual execução de um job agendado. A maioria dos motores e dos poolers permite que um cliente anexe um rótulo curto à sua conexão, normalmente chamado de application name, e ele aparece no log do banco de dados e na visão da atividade atual.

Uma convenção vale mais do que o recurso. Colocaria três coisas no rótulo: o serviço, a instância ou a execução e o release. Por exemplo, `reporting-job/run-0412/v31` diz qual job, qual execução de qual noite e qual build. O rótulo transforma “alguém está segurando quarenta conexões” em “a execução de terça-feira do relatório segura quarenta conexões, e é o build novo”.

Duas ressalvas acompanham isso. Primeiro, o rótulo é fornecido pelo cliente e pode ser definido como qualquer coisa, de modo que serve para diagnóstico e atribuição, não para autorização. O login é a identidade; o rótulo é uma anotação no envelope. Segundo, um pooler em modo de transação compartilha conexões com o servidor entre clientes, de modo que o rótulo do lado do servidor pode não ser o do cliente. Alguns poolers o repassam a cada transação, outros não, e o log do próprio pooler sobre os seus clientes passa a ser o registro que carrega o nome. Descubra o que a sua configuração faz antes de depender disso.

O rótulo também alimenta o Capítulo 14: um log de auditoria é mais fácil de ler quando cada linha tem o serviço e a execução, e não apenas um login. O custo é uma linha na inicialização de cada serviço e mais uma convenção a manter quando um novo serviço aparece.

As conexões não são o único recurso que um papel pode consumir. Memória para ordenar e juntar, espaço temporário em disco, CPU e o número de workers paralelos também são compartilhados, e os motores diferem muito quanto a conseguirem limitá-los por papel. Onde o motor não consegue, o isolamento precisa vir da colocação: um leitor pesado vai para uma réplica, que é o argumento do Capítulo 6. Verifique quais limites o seu motor consegue definir por papel antes de prometer qualquer um deles a alguém.

Ledgerly: a noite em que o relatório devorou a API

Volte à noite de terça-feira. O estado inicial a explica em uma frase: o `reporting-job` rodava como `admin` contra o `primary`, de modo que competia pelos mesmos slots que o `billing-api` sob o mesmo nome, sem limite em nenhum dos dois e sem timeout em nada.

O plano de Alex não começa pelo pooler. Começa pelo que é mais barato e dá mais proteção, e vai descendo pela lista conforme a equipe aprende.

1. **Limites por papel.** Com os papéis de aplicação do Capítulo 3 já em vigor, cada um recebe um limite de conexões a partir de uma tabela como a de cima. O relatório perde de imediato a capacidade de tomar os slots da API, mesmo que ainda leia o `primary` na primeira noite.
2. **Timeouts por papel.** `app_billing_api` recebe um `statement timeout` curto e um limite curto de `idle in transaction`, e o relatório recebe um `statement timeout` longo que não se aplica à API.
3. **Rótulos.** Cada serviço e cada execução agendada define o seu rótulo. O próximo page nomeia a execução, e não apenas o papel.
4. **Um pooler.** Os cinco serviços se conectam por ele, e ele passa a ser a única entrada, com o total por papel vindo de um dos três lugares descritos acima.
5. **Uma mudança para a réplica.** O `reporting-job` lê de `ledgerly_replica_2` com o seu próprio orçamento, e o `primary` deixa

de carregar o trabalho dele. O Capítulo 6 descreve a própria mudança.

6. **Um alerta antes da recusa.** A equipe de plataforma observa a saturação de cada pool, e não apenas as recusas. Um pool que fica em, digamos, noventa por cento do limite por horas diz “aumente o limite ou conserte o chamador” enquanto ainda há tempo.

O job também precisa de uma mudança própria, porque um limite que recusa o job só é útil se o job reage bem a ser recusado. Um job que tenta de novo imediatamente depois de uma recusa é a multidão na porta em escala menor.

```
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
```

Com isso no lugar, a mesma mudança que aumentou o paralelismo do relatório teria um resultado diferente. O job atinge o limite e é recusado, desacelera por conta própria e leva mais tempo, e a API não fica sem slots por causa disso. Um job que desiste de um lote também deve deixar um registro do que pulou, para que nada leia agregados pela metade como se estivessem completos; isso é o design do próprio job e vai além deste capítulo. A revisão que aprovou a mudança continuaria correta, porque o design deixou de depender de um revisor perceber a interação entre dois serviços.

Trade-offs e custos

Limites por papel custam ajuste. Um limite baixo demais recusa carga legítima, e a recusa parece uma indisponibilidade para o serviço que a recebe. Os limites precisam de um responsável e de uma revisão sempre que um serviço faz scale-out ou um novo workload aparece, porque a soma de instâncias vezes tamanhos de pool pode ultrapassar o limite sem que nenhuma mudança isolada pareça errada. A equipe de plataforma passa a ser o lugar onde essas perguntas são feitas, o que é útil e também é uma nova fila.

Um pooler é um novo sistema no caminho. Toda consulta agora passa por ele, e quando ele cai, todos os serviços caem. Ele precisa estar disponível durante o incidente em que você mais precisa dele, de modo que precisa de redundância, monitoramento e runbook próprios. No modo de transação, ele também remove recursos de sessão dos quais algum código depende e torna o log do banco de dados mais difícil de ler. Se você tem um único serviço pequeno com uma instância e um pool de dez, o pool da própria aplicação mais um limite por papel bastam.

Os timeouts cancelam trabalho legítimo. Um statement timeout apertado demais mata uma boa instrução, e um retry ingênuo repete o trabalho e o cancelamento. Os serviços precisam tratar o cancelamento, e para escritas isso significa pensar se a operação pode ser executada duas vezes sem dano. Os valores para `ledgerly_migrator` e para as correções de dados (Capítulo 10) diferem dos das aplicações, o que significa mais configuração para manter correta.

Um limite que ninguém observa não protege nada. Um statement timeout tão longo que nunca dispara, ou um limite de papel que nenhum alerta cobre, acrescenta uma linha à configuração e nenhum limite. O teste é se alguém perceberia caso o controle parasse de funcionar.

Rótulos são baratos e falsificáveis. Ajudam as pessoas e custam disciplina, mas não provam nada, e um log que dependa deles para atribuição depende de uma convenção.

Quando não usar estes controles: não acrescente limites por papel antes de os papéis existirem, pois um limite no papel `admin` compartilhado só limita todos de uma vez. Não defina timeouts agressivos em um serviço antes de saber quanto tempo duram as suas instruções normais, porque a primeira execução de faturas cancelada ensinará a equipe a desligar os timeouts. Meça primeiro o normal e depois defina o limite acima dele.

Erros comuns

- **Aumentar o limite do servidor para resolver um incidente.** É tentador porque funciona na hora e não custa nada hoje. Desloca a falha para uma multidão maior e remove o sinal precoce. Dê a cada papel um orçamento e deixe o limite do servidor como a soma.

- **Dimensionar o pool para o tráfego que você espera ter.** Mais conexões parecem mais capacidade, mas o banco de dados tem uma quantidade fixa de concorrência útil. Dimensione o pool pelo trabalho que uma instância consegue fazer ao mesmo tempo.
- **Mapear todos os clientes para um único login no pooler.** É a configuração fácil, e ela traz de volta a conta de administrador compartilhada. Mantenha o login próprio de cada instância ou escolha explicitamente um login estável por papel e aceite o que o log deixa de dizer.
- **Usar o modo de transação com estado de sessão.** O código funcionava no modo de sessão, então ninguém o testa no modo de transação. O contexto das regras de linha, a identidade para auditoria e as tabelas temporárias precisam viver dentro da transação, ou o modo precisa continuar no nível da sessão para esse papel.
- **Um único conjunto de timeouts para todos os papéis.** Um valor padrão único é apertado demais para o batch e frouxo demais para a API, e o desfecho usual é que ele é aumentado para todos. Associe os valores ao papel.
- **Tratar o application name como identidade.** O rótulo é definido pelo cliente. Use o login para autenticar e o rótulo para explicar.
- **Encher o servidor com orçamentos de aplicação.** Quando os limites somam o servidor inteiro, a equipe de plataforma não consegue se conectar durante o incidente. Mantenha uma reserva fora de todos os papéis de aplicação.

Pontos-chave

- Uma conexão é um recurso limitado e um ponto de política: um limite que pertence ao servidor e não a um papel deixa o primeiro chamador ganancioso levar a parte de todos.
- Um orçamento de conexões por papel transforma “o banco de dados está cheio para todos” em “este chamador ficou sem slots”, o que reduz o raio de impacto de um job descontrolado ao próprio job.
- Mantenha uma reserva de conexões fora de todos os papéis de aplicação, para que as pessoas e as ferramentas que consertam o sistema sempre consigam se conectar.

- O pooler é por onde os serviços entram, e por isso é o lugar da recusa central e do chokepoint de rede. Muitos poolers não conseguem somar os logins dinâmicos de um papel, de modo que o total por papel vem do banco de dados, do broker ou de um login único no upstream por papel, cada um com um preço nomeado, como a seção sobre o pooler apresenta.
- O pooling em modo de transação remove o estado de sessão, de modo que qualquer contexto de que dependa uma regra de linha ou um registro de auditoria precisa ser definido dentro da transação e precisa falhar de forma ruidosa quando estiver ausente.
- Os timeouts de statement, de lock wait e de idle in transaction pertencem ao papel, e o statement timeout do banco de dados deve disparar antes do deadline da aplicação. Uma sessão pode suspender um valor padrão, de modo que, para as pessoas, o limite precisa ser imposto onde elas não podem alterá-lo.
- Os application names tornam as conexões explicáveis, mas não provam nada, de modo que o login continua sendo a identidade e o rótulo continua sendo uma anotação.

Um limite de conexões e um pooler decidem quantos chamadores chegam ao primary, mas não mudam quanto trabalho cai sobre ele. O Capítulo 6 dá o próximo passo: mover as leituras para réplicas e outras cópias, e as questões de acesso que chegam com cada cópia.

Capítulo 6 — Leituras, réplicas e cópias de dados

Imagine uma terça-feira de manhã na Ledgerly, poucos dias antes do fim do mês. Pediram a Jordan, da equipe de dados, um número: quantas faturas foram emitidas no mês passado, divididas por segmento de cliente e por haver ou não uma tentativa de pagamento que já tinha falhado. O lugar natural para perguntar seria a cópia de analytics, mas ela é uma restauração noturna, e a restauração tem um dia de idade. A pergunta exige os dados de hoje. Jordan abre um cliente desktop, conecta-se com o login de admin compartilhado e executa uma consulta que une `invoices`, `invoice_lines` e `payments` ao longo do mês inteiro.

Nada na consulta está errado. É uma pergunta razoável, escrita por alguém competente no que faz. Mas a conexão aponta para o `primary`, porque esse é o host de que o cliente se lembrava, e a consulta lê várias tabelas grandes enquanto o `billing-api` atende clientes que estão prestes a pagar suas faturas. O aplicativo web fica lento, enviam um page para Robin, e Robin e Jordan passam vinte minutos até descobrir que o relatório e a lentidão são o mesmo problema. Ninguém fez nada fora do comum. O design oferecia um único lugar para ler, e toda leitura ia para lá.

Este capítulo acrescenta um passo ao estado-alvo: as leituras vão para réplicas, e os relatórios nunca tocam o `primary`. No modelo de cinco partes do Capítulo 1, trata-se do Grant e do Caminho trabalhando juntos, que é o que autorização significa neste livro. O grant decide o que quem lê enxerga, e o caminho decide a que banco de dados essa pessoa consegue chegar. O capítulo trata também do que decorre desse passo: cada réplica, e cada outra cópia dos dados, é um novo lugar em que o acesso precisa ser projetado. A pergunta é esta: quando os dados são lidos em mais lugares do que são escritos, quais leituras vão para onde, quem pode ler cada cópia e como evitar que as cópias se multipliquem em portas que ninguém possui?

O problema e por que ele é difícil

Um banco de dados que recebe escritas e também atende a todas as leituras está fazendo dois trabalhos com um único conjunto de recursos. As escritas precisam do **primary**: há apenas um, ele é a fonte da verdade e precisa continuar disponível para elas. As leituras podem ser repetidas, podem ser distribuídas, e muitas toleram dados com alguns segundos de atraso. Tirar as leituras do **primary** protege as escritas, e proteger as escritas é proteger o livro-razão.

A abordagem óbvia é adicionar uma réplica e apontar para ela as cargas que mais leem. Isso funciona por cerca de uma semana. Depois disso, surgem três problemas, e nenhum deles é resolvido pela réplica em si.

O primeiro é que uma réplica não é o **primary**. Na maioria das implantações, a replicação é assíncrona: o **primary** faz o **commit** de uma alteração, e a réplica a aplica pouco depois. A diferença se chama **replica lag**. Uma leitura que cai em uma réplica atrasada devolve uma resposta verdadeira sobre um mundo um pouco mais antigo. Para um relatório mensal, isso é invisível. Para um cliente que acabou de pagar uma fatura e olha uma página que ainda diz “em aberto”, é um ticket de suporte.

O segundo problema é que os privilégios viajam com os dados. Uma réplica física é uma cópia completa, inclusive das colunas **Restricted**, e qualquer coisa que consiga ler a réplica consegue ler o que quem lê o **primary** conseguiria, a menos que os **grants** digam o contrário. A próxima seção explica por que um papel não pode ser tornado mais seguro em uma única réplica.

O terceiro problema é que as cópias se multiplicam. A primeira réplica é para a aplicação. A segunda é para relatórios, depois para investigação humana, depois para a etapa de **export**. Segue-se um **warehouse**, alimentado por um agendamento ou por **change data capture**. Alguém restaura um **snapshot** em um banco de dados de **staging**. Cada cópia foi uma resposta sensata a um problema real de carga ou de conveniência em seu dia. Cada uma também guarda dados de produção, tem suas próprias credenciais e seu próprio caminho de rede, e raramente consta da lista de caminhos de entrada para os dados de alguém. Uma **porta lateral** é uma cópia de da-

dos ou um caminho de export que contorna os controles de acesso do primary. Este capítulo trata das cópias que você constrói de propósito, e o Capítulo 13 volta às que surgem por acidente.

Uma política que diz “não execute relatórios no primary” não ajuda Jordan. O cliente se lembra de um único host, o login funciona em todos eles, e o primary é o que sempre tem dados novos. O controle precisa estar na forma como as leituras são roteadas e naquilo a que cada login consegue se conectar.

Dividindo o caminho de leitura

Começaria tratando o caminho de leitura como uma superfície de acesso própria, com inventário próprio, papéis próprios e regras próprias, e fazendo uma pergunta a cada leitura: ela precisa do primary? Uma leitura precisa dele quando decide algo, como um saldo antes de um pagamento ser cobrado, ou quando precisa enxergar uma escrita feita um instante antes na mesma sessão. Todo o resto pode ir para uma réplica. Como um serviço classifica e roteia suas próprias leituras é design de aplicação, e este livro o deixa de lado.

A regra de acesso é a outra metade, e tem duas linhas. As leituras de relatórios e as leituras humanas nunca recorrem ao primary: se a réplica estiver indisponível, o job espera ou falha, e a equipe de plataforma é avisada. E os logins usados para elas não conseguem se conectar ao primary de forma alguma, de modo que um erro não possa enviá-los para lá. Um agregado noturno que roda duas horas atrasado é um inconveniente. Um que passa discretamente para o primary porque a réplica estava fora do ar é como o estado inicial surgiu.

Um papel somente leitura não é inofensivo

As leituras costumam ser tratadas como carga. Elas também são a principal exposição de confidencialidade: um conjunto de resultados que sai do sistema não altera nada, não deixa nada para reverter e pode ser copiado para qualquer lugar. Um papel que não consegue escrever ainda consegue ler todas as colunas que lhe foram concedidas, e uma única consulta pesada dele pode derrubar o primary. “Somente leitura” também significa três mecanismos diferentes, que variam quanto a a sessão poder desfazê-los.

Mecanismo	O que impede	A sessão pode desfazê-lo?
Um grant sem privilégios de escrita	Escritas feitas por esse papel	Não por si só; mas uma sessão que consegue mudar para um papel com privilégios de escrita deixa de estar presa a ele
Um padrão de transação somente leitura	Escritas nessa sessão, por padrão	Em muitos motores, sim: a sessão pode revertê-lo ou iniciar uma transação de leitura e escrita
Uma réplica física	Toda escrita nesse host	Não: o host recusa escritas, e a sessão não tem configuração alguma para mudar isso

Para as pessoas, a réplica é o mais forte dos três, e esse é um dos motivos pelos quais as leituras humanas rotineiras vão para ela. O que nenhum deles limita é quanto uma sessão pode ler. Um row cap em uma sessão e a divisão em níveis para exports pertencem à Execução, e o Capítulo 9 trata deles.

Os grants viajam para a réplica

A pergunta sobre a réplica que os revisores de segurança fazem primeiro é “o que alguém pode fazer nela?”. Em motores que replicam fisicamente, os papéis e os grants vivem no catálogo do cluster, e esse catálogo é copiado junto com os dados. Um papel é definido uma vez, no primary, e aparece em todas as réplicas. Uma réplica é somente leitura, então um papel não pode ser criado, alterado nem concedido apenas nela. Daí decorre que um papel não pode ser tornado mais seguro em uma única réplica: estreitar a superfície de quem lê em uma réplica exige a camada de conexão ou uma cópia separada. Os privilégios que você projetou no Capítulo 3 valem em todas as réplicas, e a que banco de dados um login pode chegar é uma regra da camada de conexão.

Isso tem um lado útil. O `human_readonly` retém as colunas Restricted e mascara as Confidential onde o Capítulo 12 manda, e faz isso na réplica pelo mesmo motivo que faz no primary. O `app_reporting_job` lê em uma réplica sem acesso às colunas Restricted. Se os grants foram construídos com o privilégio mínimo em mente, uma réplica não dá a quem lê nada além do que o papel permite.

Tem também um lado arriscado. No estado inicial, as réplicas aceitam o login de admin compartilhado, que é proprietário de todas as tabelas. Quem lê em uma réplica com esse login vê todas as colunas, o que é a mesma exposição do primary menos a capacidade de escrever. As pessoas tendem a tratar as réplicas como seguras exatamente por esse motivo, e o resultado é que as réplicas são onde mora o acesso mais amplo.

Apontando cada login para a sua réplica

O estado-alvo dá a cada réplica uma finalidade, e essa finalidade é imposta onde as conexões são aceitas. Dois termos são usados em sentidos fixos daqui em diante. Uma **network policy** diz quais hosts podem abrir uma conexão TCP com um host de banco de dados; o Capítulo 7 escreve uma. Uma **connection policy** diz qual login pode se autenticar em qual banco de dados, e de onde.

A connection policy precisa de um mecanismo, porque os grants não conseguem carregá-la e a network policy não enxerga logins. Nos motores que têm um, o mecanismo é a regra de acesso baseada em host que cada servidor lê da própria configuração. Essa regra se baseia na origem da conexão e em um padrão de login ou papel, e não faz parte dos dados replicados, de modo que o primary e cada réplica podem ter regras diferentes. O bloco abaixo é uma dessas políticas, escrita como as regras que os três servidores teriam.

```
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
```

O custo é que agora três conjuntos de regras precisam ser mantidos em sincronia com o esquema de papéis, e uma alteração que chega a um servidor e não a outro é drift. Os conjuntos de regras pertencem ao mesmo pipeline que aplica as definições de papéis (Capítulo 3), e uma verificação que compare as regras de cada servidor com a política pertence à mesma revisão. Onde um motor não tem uma regra desse tipo, as mesmas decisões recaem sobre o pooler e o gateway, e sobre a network policy que os admite.

O pareamento de `human_readonly` com `ledgerly_replica_2` significa que uma pessoa que investiga um problema lê da mesma réplica que os relatórios usam. É um trade-off deliberado: uma réplica para proteger, monitorar e dimensionar, em troca de a investigação humana competir com o job noturno por capacidade. Se os dois começarem a interferir, uma terceira réplica para as pessoas é uma divisão sensata, e a connection policy muda em um bloco.

As negações no primary importam tanto quanto as permissões nas réplicas. O `app_reporting_job` e o `app_vendor_export` não têm o que fazer no primary. Se seus logins não conseguem se conectar lá, o comportamento de “às vezes lê o primary” do estado inicial deixa de ser algo que um erro de configuração possa trazer de volta. Um job apontado para o host errado falha ao conectar, o que é uma falha visível e barata.

A exceção no sentido contrário é uma sessão JIT ou break-glass. Todo login de sessão humana é membro de `human_readonly` por padrão, de modo que a regra do primary não pode ser escrita contra esse papel: ela ou cortaria o break-glass ou não significaria nada. Ela é escrita contra o tipo de sessão que o access gateway abriu. Uma sessão rotineira é enviada para `ledgerly_replica_2`. Uma sessão que precisa escrever, ou ler em uma emergência o que uma réplica não consegue mostrar com rapidez suficiente, é aberta no primary, e a requisição de acesso é feita a partir dessa sessão (os capítulos 7, 8 e 10 descrevem o modelo).

Isso deixa uma janela no primary: até que uma requisição seja aprovada, uma sessão assim pode fazer o que `human_readonly` faz. Três regras mantêm a janela pequena, e o Capítulo 7 as enuncia como parte dos tipos de sessão. Só podem abrir esse tipo as pessoas a quem a política permite solicitá-lo. O motivo é registrado em log quando a sessão se abre, e a sessão tem uma curta janela de abertura na qual uma requisição precisa ser feita.

Um alerta dispara se nenhuma for feita. Quem só quer olhar os dados não abre uma sessão assim; é para isso que serve a réplica, e é por isso que as leituras rotineiras não são permitidas no primary.

A réplica também tem um host

Uma réplica física é uma cópia dos arquivos de dados, de modo que seus hosts, volumes, snapshots e credenciais de replicação chegam aos mesmos dados sem passar por um papel de banco de dados. Aplicaria às réplicas as mesmas regras de acesso ao host, a mesma criptografia de armazenamento e a mesma retenção do primary, e registraria isso por escrito quando a réplica fosse criada, porque ninguém vai se lembrar depois. O Capítulo 13 volta aos snapshots e aos backups.

Relatórios, analytics e a escolha da cópia

As réplicas resolvem um problema de carga para leituras baratas e frequentes. Os relatórios são o oposto: leituras raras, grandes e lentas. A questão é se os relatórios pertencem a uma réplica ou a outra coisa, e a resposta depende do que os relatórios precisam.

Três lugares onde um relatório pode ler

Opção	Atualidade	Carga no primary	Superfície de acesso	Adequada para
Uma réplica de relatórios (ledgerly_replica_2)	De segundos a minutos	Apenas a replicação	Os mesmos dados e papéis do primary	Relatórios operacionais, exports, investigação
Uma cópia separada, carregada em um agendamento (ledgerly_analytics)	Tão antiga quanto a última carga	Uma carga por agendamento	Papéis próprios e dados próprios	Modelos, dashboards, análise em massa pela equipe de dados
Uma cópia alimentada por change data capture	De segundos a minutos	Um leitor do stream de alterações	Um novo pipeline e um novo armazenamento	Analytics quase em tempo real, muitos consumidores

Enviaria os relatórios operacionais para uma réplica e a análise em massa para uma cópia separada, e resistiria a pôr o trabalho da equipe de dados na réplica por um motivo que não tem a ver com carga. Uma réplica carrega a forma e a sensibilidade do esquema de produção. A equipe de dados não precisa das colunas Restricted e, no estado-alvo, também não vê as colunas Confidential em bruto, porque a cópia de analytics é mascarada à medida que é carregada, exceto pelas dicas de exibição que existem para serem mostradas (o Capítulo 12 descreve como e nomeia a exceção). Uma réplica não pode ser mascarada por consumidor sem mudar o que a réplica é. Uma cópia separada pode ser construída para o seu público.

É também por isso que a cópia de analytics não é chamada de réplica em nenhum ponto deste livro. Ela é carregada em um agendamento a partir do primary, tem papéis diferentes, e seu conteúdo pode diferir do da origem. Chamá-la de réplica convida a supor que ela carrega os mesmos controles, e no estado inicial ela carrega menos.

O change data capture é a opção a que recorreria por último. Ele acrescenta um pipeline que precisa ser mantido em funcionamento, um armazenamento que precisa ser protegido, um atraso próprio e um contrato de esquema entre produtor e consumidor. Para uma empresa do tamanho da Ledgerly, adotaria primeiro a carga agendada e registraria por escrito o que me faria mudar de ideia: por exemplo, relatórios que precisam de dados mais novos do que o agendamento permite, ou uma segunda equipe com a mesma necessidade.

O custo de cada cópia

Toda opção da tabela tem o mesmo item oculto: algo novo para proteger. Uma cópia é um lugar onde os dados vivem, um conjunto de logs que conseguem lê-los, um caminho de rede, uma regra de retenção e um responsável. Uma cópia sem responsável é uma porta lateral, e uma cópia com responsável e sem expiração é uma porta lateral que demora mais para ser notada.

Ajuda manter um registro curto das cópias, com os mesmos campos para todas. O bloco abaixo é uma notação ilustrativa, e o ponto dele é que as linhas `owner` e `ends` não podem ficar em branco.

```

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

```

Para a Ledgerly, o registro começa com cinco entradas: as duas réplicas, a cópia de analytics, os arquivos de export que vão para o fornecedor de contabilidade e os dumps antigos no armazenamento compartilhado. Quatro são de propósito e os dumps não são, e é por isso que o registro é útil: os dumps ganham um responsável e uma data de término próxima. O Capítulo 13 usa o mesmo registro para as cópias que vivem fora dos bancos de dados.

Nota: Um registro que ninguém abre não é um controle, por isso ele é amarrado à etapa que cria cópias. A etapa de provisionamento se recusa a criar uma réplica, uma restauração de snapshot, um banco de dados de staging ou um job de export a menos que exista uma linha com um responsável e, onde o tipo exigir, uma data de término. A revisão de acessos (Capítulo 14) então pergunta quais cópias existem e quem pode ler cada uma.

A recusa tem um custo: alguém preenche uma linha antes de uma equipe com prazo conseguir sua cópia, e as primeiras semanas geram reclamações. Ela também cobre apenas as cópias feitas pela etapa de provisionamento, de modo que uma cópia feita à mão continua sendo uma porta lateral, e o Capítulo 13 trata disso.

Alcance, e não só grants

Um papel que está correto no papel continua errado se o login errado consegue se conectar. Para cada cópia, o grant retém o que o público não precisa (capítulos 3 e 12), e o alcance decide quais caminhos de rede e quais

principais conseguem abrir uma conexão. Em uma rede plana, um login vazado em qualquer serviço chega a todas as cópias, e o alcance limita o que vale um vazamento, como diz o Capítulo 4. A tabela enuncia a regra de alcance para cada lugar; a *network policy* que impõe o lado do host é a do Capítulo 7, e a *connection policy* acima impõe o lado do login.

Lugar	Alcançável a partir de	Não deve ser alcançável a partir de
ledgerly_primary	Hosts do pooler, o gateway para sessões JIT e break-glass, a etapa do pipeline que aplica migrações	Jobs de relatórios e de export, sessões humanas rotineiras, a camada web diretamente
ledgerly_replica_1	Hosts do pooler, para os serviços que fazem navegação	O gateway para leituras rotineiras, a carga de analytics, a camada web diretamente
ledgerly_replica_2	Hosts do pooler para relatórios e export, o gateway para leituras humanas rotineiras	Serviços que executam a aplicação, a camada web diretamente
ledgerly_analytics	O job de carga, o reporting-job para seus resultados, o gateway para sessões de analytics	Os serviços que executam a aplicação

Migrando o caminho de leitura da Ledgerly

Considere como Alex levaria a Ledgerly do estado inicial para este design sem quebrar o job noturno na primeira tentativa. A ordem importa mais do que a velocidade, e a seguiria em cinco passos.

- 1. Inventariar as leituras.** Para cada serviço, liste as consultas que ele executa e separe-as em leituras que precisam do primary e leituras que não precisam. A maioria das consultas de listagem do `billing-api` é do segundo tipo. Suas escritas e seus redirecionamentos pós-escrita não são. Os logs de consultas são a matéria-prima das regras de roteamento.
- 2. Dar a cada réplica uma finalidade.** Escreva a *connection policy* de `ledgerly_replica_1` e `ledgerly_replica_2` e aplique-a primeiro aos novos logins, deixando o login compartilhado no lugar por enquanto.
- 3. Mover o `reporting-job` e a etapa de export.** Aponte os dois para `ledgerly_replica_2` com logins próprios e, em seguida, negue esses

logins no primary. Execute-os lado a lado por alguns ciclos e compare as saídas antes de desligar o caminho antigo.

4. **Mover as leituras de navegação.** Roteie as leituras de navegação do `billing-api` e do `admin-console` para `ledgerly_replica_1` atrás de uma flag, e aumente a parcela aos poucos.
5. **Fechar o login compartilhado nas réplicas.** Quando nada legítimo o usar ali, remova sua capacidade de se conectar a qualquer uma das duas réplicas. Este é o passo que torna o resto permanente, e é o que as equipes adiam.

O caminho da equipe de dados também muda. A pergunta de fim de mês de Jordan agora tem duas respostas. Se um número com um dia de idade serve, ela vai para `ledgerly_analytics` por meio de `human_analytics_reader`. Se precisa dos dados de hoje, vai para `ledgerly_replica_2` pelo access gateway com `human_readonly`, uma sessão que é registrada em log, que não enxerga as colunas Restricted e que não consegue tocar o primary. O que Jordan perde é a capacidade de apontar um cliente desktop para qualquer host e ver funcionar. O que Jordan ganha é que uma consulta pesada passa a ser um problema para a réplica e um problema pequeno para todos os outros.

Protegendo uma réplica de quem a lê

Uma réplica que atende pessoas e um job noturno vai receber consultas pesadas, e em alguns motores uma consulta longa pode atrasar a replicação ou ser cancelada para que a replicação continue, dependendo da configuração. Os controles são os do Capítulo 5, aplicados aos papéis da réplica: um statement timeout mais curto para `human_readonly` do que para `app_reporting_job`, um limite de conexões por login que o gateway define em cada login de sessão humana e uma etapa de export que roda fora da janela do agregado noturno. Acrescente um alerta de lag e envie o page para quem for o responsável pela réplica no registro. Um alerta que ninguém recebe não é um controle.

Trade-offs e custos

Uma réplica é um sistema a operar. Ela precisa ser monitorada, atualizada, dimensionada e recuperada, e para uma equipe pequena uma réplica e uma cópia noturna podem ser tudo o que é necessário. O argumento de acesso continua valendo sem ela: você decide quais logins chegam ao primary e quais não chegam.

Compartilhar a `ledgerly_replica_2` tem um preço. As pessoas e o job noturno competem pela capacidade de uma réplica, em troca de um único lugar para proteger. A divisão em uma terceira réplica é barata em termos de política e cara em operação.

A connection policy é mais configuração. As regras por servidor precisam corresponder ao esquema de papéis e sobreviver a um failover, a uma reconstrução e a uma nova réplica. O drift entre servidores é a falha, e a verificação de comparação é o preço de mantê-lo visível.

Algumas telas ficarão atrasadas. Tirar as leituras do primary significa lag, e alguém vai notar. Combine antecipadamente com o suporte e o produto quais telas podem ficar atrasadas e verifique o lag primeiro quando algo não fechar em uma investigação em uma réplica (Capítulo 15).

O registro acrescenta atrito. Exigir uma linha antes de a cópia existir atrasa a equipe que queria a cópia, e esse atraso é o efeito pretendido. Um registro que ninguém precisa preencher na criação é uma lista do que alguém lembrou.

Há casos em que o conselho deste capítulo não se aplica. Um sistema pequeno, cujo primary tem folga e em que os engenheiros que leem dele já se conectam por um caminho controlado, pode não precisar de uma réplica por causa de carga, e separar o caminho de leitura tem um retorno menor. Um sistema com exigências de atualidade muito estritas para toda leitura talvez tenha de executar a maioria das leituras no primary e concentrar o esforço em limitar o que essas pessoas podem fazer. Uma réplica também não é um backup; o Capítulo 13 trata dos backups.

Erros comuns

- **Tratar uma réplica como menos sensível do que o primary.** Como não se pode escrever nela, ela acumula acesso amplo: o login compar-

tilhado, um papel de leitura permanente para todos, uma regra de firewall mais larga. Ela guarda os mesmos dados, então recebe os mesmos grants e uma connection policy igualmente estreita.

- **Deixar os relatórios ou as leituras humanas recorrerem ao primary.** Recorrer a ele é tentador porque mantém o relatório rodando quando a réplica está fora do ar. Faça a leitura falhar e negue esses logs no primary, para que um erro não possa enviá-los para lá.
- **Enviar para uma réplica leituras que decidem algo.** A leitura é barata e a réplica está ociosa, então parece uma vitória fácil. Uma decisão tomada com dados desatualizados e seguida de uma escrita é como saldos e status saem de sincronia. Classifique as leituras pelo que vem depois delas, não pelo quanto pesam.
- **Montar um warehouse apontando uma pessoa para uma réplica.** A equipe de dados passa então a carregar o esquema de produção, a sensibilidade da produção e os limites de capacidade da produção. Dê ao analytics uma cópia própria e papéis próprios (capítulos 12 e 13).
- **Adicionar cópias sem um registro.** Cada uma foi um favor a uma equipe com prazo. Seis meses depois, ninguém sabe dizer quantas existem. Faça do responsável e da data de término parte da criação de uma cópia.
- **Esquecer o host e os snapshots.** Os papéis estão certos, e o snapshot de volume da réplica pode ser lido por qualquer pessoa com acesso à infraestrutura. Inclua o host, o armazenamento e os snapshots nos controles de cada cópia.
- **Supor que um papel pode ser restringido por réplica.** Em motores que replicam fisicamente, os papéis valem para o cluster inteiro. Estreite a superfície de quem lê na camada de conexão ou com uma cópia separada.

Pontos-chave

- As escritas precisam do primary e a maioria das leituras não, então a primeira pergunta para o caminho de leitura é se uma leitura decide algo ou segue uma escrita; as leituras de relatórios e as leituras humanas nunca recorrem ao primary.

- “Somente leitura” são três mecanismos, e só a réplica não pode ser desfeita por uma sessão, de modo que um papel somente leitura limita as escritas e não o que uma sessão pode ler.
- Uma réplica guarda os mesmos dados do primary e é regida pelos mesmos papéis válidos para o cluster inteiro, de modo que os grants do Capítulo 3 e a connection policy decidem em conjunto o que quem lê pode ver.
- A connection policy é uma regra por servidor, baseada na origem e no login, mantida em sincronia com o esquema de papéis; cada réplica tem uma finalidade, e o login de admin compartilhado não é aceito em nenhuma delas.
- Uma sessão JIT ou break-glass no primary exige uma pessoa autorizada a abri-la, um motivo registrado em log, uma janela curta e um alerta quando nenhuma requisição se segue.
- O analytics pertence a uma cópia separada, construída para o seu público, e chamá-la de réplica esconde o fato de que ela carrega controles diferentes.
- Toda cópia de dados precisa de um responsável, de uma lista de quem pode lê-la e, exceto para as réplicas e a cópia de analytics, de uma data de término, e a etapa que cria cópias se recusa a rodar sem eles.

O caminho de leitura agora tem papéis próprios, destinos próprios e uma lista curta de cópias, mas ainda não dissemos como uma pessoa chega a qualquer um deles. O próximo capítulo trata dos caminhos de entrada para as pessoas e compara um bastion, um proxy de banco de dados e um access gateway.

PARTE III

Acesso humano

Capítulo 7 — Caminhos de entrada: bastions, proxies e access gateways

Imagine que Alex, da equipe de plataforma, ouve uma pergunta simples durante uma reunião de planejamento: “Se desligássemos amanhã um caminho de entrada para o banco de dados de produção, qual deles doeria?”. Ninguém responde rápido. Alex vai descobrir e começa pela rede. A VPN dá a cada engenheiro uma rota até o primary. Há também um bastion host que alguém montou para um prestador de serviço dois anos atrás, e ele ainda tem uma porta escutando. Os clientes desktop de uma dúzia de laptops guardam conexões salvas; uma delas foi configurada para passar por um túnel pelo bastion, enquanto outra vai direto pela VPN.

Depois Alex pergunta a Robin, que está de on-call esta semana, como Robin chegaria ao banco de dados à noite. Robin cita dois caminhos sem hesitar e um terceiro depois de pensar. Cada caminho funciona, nenhum foi projetado, e todos terminam no mesmo login de admin compartilhado. A equipe de rede consegue dizer quem está na VPN. O banco de dados consegue dizer que o login de admin se conectou. Ninguém consegue dizer qual pessoa estava por trás de uma determinada conexão, e ninguém consegue desligar o acesso de uma pessoa sem desligar o de todas.

Assim, a pergunta deste capítulo é: o que deve existir entre uma pessoa e o banco de dados, para que alcançar a rede e alcançar os dados sejam duas decisões separadas, cada uma com um nome associado? Este capítulo acrescenta o access gateway, o primeiro passo do lado humano do estado-alvo da Ledgerly. Nas cinco partes do livro, ele carrega o Caminho e a Identidade de quem entra por ele: a rede decide quem pode bater à porta, e o gateway põe um nome em cada sessão. Os papéis que uma sessão do gateway recebe, e como uma pessoa pede mais do que leitura, pertencem ao Capítulo 8.

O problema e por que ele é difícil

Um caminho de rede e um caminho de banco de dados parecem a mesma coisa, porque o primeiro é o que você precisa antes de poder tentar o segundo. Uma pessoa que está na VPN pode abrir uma conexão com a porta do banco de dados. Se essa pessoa pode então fazer alguma coisa é decidido por um nome de usuário e uma senha, e no estado inicial da Ledgerly a senha é compartilhada. A rede disse “você pode bater à porta”, e o banco de dados não conseguiu saber quem bateu.

Esse é o cerne do problema. Os controles de rede respondem à pergunta “quais máquinas podem falar com quais máquinas”, e o fazem bem. Eles não respondem a “que pessoa é esta, o que ela tem permissão de fazer agora e o que ela fez”. Quando se pede a uma regra de rede que sirva de substituta para essas perguntas, ela falha de forma silenciosa: a regra continua correta enquanto o acesso que ela concede se torna obsoleto. Uma pessoa que mudou de equipe continua na VPN. Um prestador de serviço cujo contrato terminou continua em um grupo que ninguém revisa.

Mais três coisas tornam o caminho humano mais difícil do que o caminho dos serviços.

Primeiro, as pessoas são os clientes mais variados. Um serviço se conecta de um lugar conhecido com um driver conhecido. Uma pessoa pode usar um cliente de linha de comando, uma GUI desktop, um notebook, uma ferramenta no navegador ou um script que escreveu na terça passada. Cada um tem seu jeito de lembrar credenciais, seu padrão de auto-commit e seu hábito de baixar resultados para o disco local. O Capítulo 1 listou os clientes que lembram a senha e fazem auto-commit; essa não é uma configuração incomum, é o comportamento padrão de muitas ferramentas feitas para a conveniência.

Segundo, as pessoas se conectam dos lugares menos controlados. Um laptop viaja, entra em repouso, passa por backup e às vezes é compartilhado em casa. Uma conexão salva com uma senha dentro se move com o laptop, e o mesmo vale para qualquer conjunto de resultados exportado para um arquivo. O banco de dados não enxerga nada disso.

Terceiro, o caminho é usado sob pressão. Quem precisa do banco de dados à noite não está com disposição para aprender uma ferramenta nova, e um caminho lento ou frágil será contornado por quem tiver os meios. Um

design correto no papel que perde para o antigo atalho às 3 da manhã não mudou nada, e é por isso que o atalho precisa ser removido, e não apenas desencorajado.

Uma política que diz “não compartilhe o login de admin” não pode ser verificada, porque o log do banco de dados mostra o login de admin e mais nada. Um design em que cada sessão tem o seu próprio login pode ser verificado, e o mesmo vale para uma regra de rede que permite que só uma coisa alcance o banco de dados. O restante deste capítulo compara as formas de construir um design assim.

Quatro caminhos de entrada, comparados

Antes da comparação, dois termos, porque as palavras são usadas de forma frouxa no setor e o livro precisa fixá-las.

Um **bastion** é um host reforçado que é o único caminho de rede para um sistema protegido. Ele encaminha conexões. Ele não entende o protocolo do banco de dados, então não distingue um **SELECT** de um **DROP**, e não sabe quem está por trás da conexão, a menos que a pessoa tenha feito login no próprio bastion.

Um **access gateway** é um intermediário consciente de identidade que autentica uma pessoa, emite uma credencial de curta duração e registra a sessão. Um **proxy de banco de dados** é outra coisa: um intermediário que entende o protocolo e que faz pooling, roteamento ou filtragem de conexões de banco de dados. Um gateway pode ser construído sobre um proxy de banco de dados, mas os dois respondem a perguntas diferentes. Um proxy conhece o protocolo; um gateway conhece a pessoa. Este capítulo usa cada termo apenas nesse sentido.

Aqui estão os caminhos que uma equipe costuma considerar, com o que cada um sabe sobre a pessoa e o que deixa para trás. Os quatro primeiros põem uma pessoa em uma sessão; os dois últimos a mantêm fora de uma.

Caminho	O que sabe sobre a pessoa	O que o log do banco de dados mostra	Principal custo
Endpoint público	Nada além do login do banco de dados	Um login, vindo de qualquer endereço	Expõe a porta do banco de dados à internet inteira

Caminho	O que sabe sobre a pessoa	O que o log do banco de dados mostra	Principal custo
VPN até o banco de dados	Que a pessoa está na rede	O login compartilhado ou por pessoa	A participação na rede é tratada como identidade
Bastion	Quem fez login no bastion host, se as contas forem por pessoa	O login que a pessoa digitou, vindo do endereço do bastion	Mais um host para atualizar, e ele não vê instruções
Access gateway	A identidade vinda do single sign-on da empresa	Um login de sessão que corresponde a uma pessoa e a uma sessão	Um sistema novo que precisa estar disponível quando alguém precisa
Caminho de produto, sem SQL	O papel da pessoa em uma ferramenta interna	O login do próprio serviço, com a pessoa na trilha de auditoria da ferramenta	Cada tarefa exige uma funcionalidade construída; perguntas livres não têm caminho
Executor de envio e execução	A pessoa que enviou e o revisor que aprovou	O login do executor, executando um conjunto de instruções que foi revisado	Toda tarefa espera por revisão; nada é interativo

As duas últimas linhas não são formas de alcançar um banco de dados. A pessoa nunca mantém uma sessão, e a instrução roda a partir de um componente que a verificou antes. Elas pertencem à comparação porque, para muitas tarefas, são a melhor resposta: uma pessoa que nunca abre uma sessão não pode deixar uma aberta. O Capítulo 1 apresentou a gama desses padrões, o Capítulo 9 trata de como o gateway verifica uma instrução e o Capítulo 10 trata dos padrões de correção de dados; este capítulo trata das sessões que restam depois que as tarefas que se encaixam nesses padrões saíram.

O endpoint público

Não exporia o banco de dados à internet pública para uso humano, e o argumento é curto. Uma porta alcançável convida tráfego automatizado que nada tem a ver com as pessoas da Ledgerly, e toda credencial desse banco de dados passa a valer a pena ser adivinhada. A única defesa que sobra é a força do login, e a premissa do livro é que logins vazam.

Há casos restritos em que um banco de dados gerenciado só é alcançável publicamente e o provedor oferece uma allowlist de endereços de origem. Se você está nessa posição, trate a allowlist como o mínimo e passe para um caminho privado assim que o provedor permitir. O teste é simples: alguém que nunca trabalhou para você poderia abrir uma conexão com a porta? Se sim, todos os outros controles deste livro partem de um ponto pior.

A VPN

Uma VPN é uma camada de rede razoável. Ela restringe quem alcança o banco de dados às pessoas que receberam uma rota, e a maioria das empresas já tem uma. O limite dela é o que ela comprova. Ela comprova que uma pessoa, ou o laptop de uma pessoa, passou nas verificações da VPN. Ela não diz que login de banco de dados a pessoa usou depois, e não registra instruções.

No estado inicial da Ledgerly, a VPN é todo o controle. Todos os que estão nela alcançam o primary, e o login de admin é o mesmo para todos. Manteria a VPN, mas lhe daria uma tarefa menor: ela leva as pessoas ao gateway e às demais ferramentas internas, e não as leva mais à porta do banco de dados.

O bastion

Um bastion acrescenta uma propriedade útil: um único lugar onde a porta da rede pode ser fechada. O firewall do banco de dados admite apenas o endereço do bastion, então desligar uma conta do bastion desliga o caminho de rede de uma pessoa. Se o bastion tem contas por pessoa e registra quem fez login nele, você ganha um registro aproximado de quem estava conectado e quando.

O que um bastion não dá é qualquer coisa abaixo da camada de rede. Quando uma pessoa tem um túnel aberto por ele, a conexão com o banco de dados é o que o cliente da pessoa fizer dela. O login ainda é digitado pela pessoa ou salvo no cliente dela. As instruções são invisíveis para o bastion. Assim, um bastion arruma o quadro da rede sem consertar o quadro da identidade, e é por isso que uma equipe pode ter um e ainda assim ser incapaz de responder “quem executou essa instrução?”.

Um bastion também tem um problema silencioso de manutenção. É um host com um shell, então precisa de atualizações, de gestão de chaves e de um responsável que se lembre de que ele existe. O host do prestador de serviço na abertura é a forma habitual dessa falha: estava correto no dia em que foi construído, e ninguém foi designado para cuidar dele depois.

O access gateway

O access gateway é para onde enviaria as pessoas, pelos motivos a seguir.

Ele autentica a pessoa no single sign-on da empresa, de modo que a identidade é a mesma que a empresa já controla. Sair da empresa, mudar de equipe e falhar em uma verificação de múltiplos fatores têm efeito no gateway sem que ninguém edite o banco de dados. Ele emite uma credencial que pertence a uma sessão e morre com ela, de modo que não há senha para salvar. E registra a sessão, de modo que “quem executou essa instrução?” tem resposta em dois lugares: o registro de sessão do gateway e o log do banco de dados, que nomeia um login correspondente a exatamente uma pessoa.

Os gateways vêm em dois formatos gerais. Alguns ficam no caminho dos dados: o cliente da pessoa se conecta ao gateway, e o gateway abre a conexão com o banco de dados em nome da pessoa. Outros apenas intermediam a credencial e saem do caminho, de modo que o cliente se conecta diretamente ao banco de dados com o login de curta duração. O primeiro formato pode registrar instruções e impor regras sobre a conexão. O segundo é mais simples e coloca menos coisas no caminho, mas então a rede precisa permitir que as pessoas alcancem a porta do banco de dados, o que traz de volta parte do problema com que este capítulo começou.

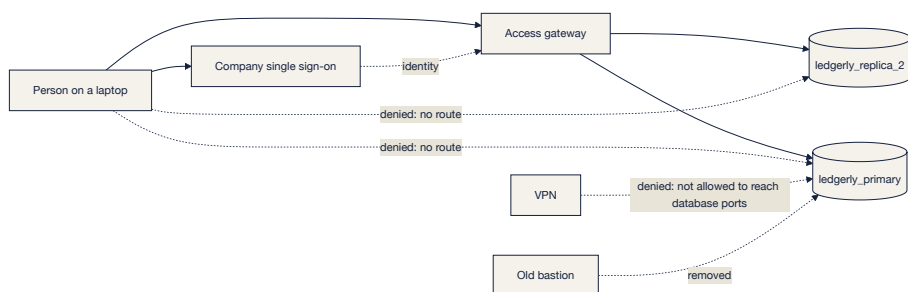
Para a Ledgerly, escolheria o formato no caminho dos dados, porque ele permite que a regra de rede seja absoluta: apenas o endereço do gateway pode alcançar os hosts do banco de dados. O preço é que o gateway trata toda instrução humana, então sua capacidade e sua disponibilidade importam. Volto a isso adiante.

Separando o alcance de rede do acesso ao banco de dados

A mudança mais útil deste capítulo não é um produto. É separar duas perguntas que o estado inicial responde juntas.

A primeira pergunta é o **alcance de rede**: quais máquinas podem abrir uma conexão com qual host e porta de banco de dados. A segunda é o **acesso ao banco de dados**: qual login pode fazer o quê, depois de conectado. No estado inicial, a VPN responde à primeira e uma senha compartilhada responde à segunda. Depois da mudança, uma regra de rede responde à primeira para todos de uma vez, e o gateway responde à segunda para cada pessoa separadamente.

O diagrama mostra o arranjo-alvo para as pessoas. Observe as setas negadas: elas são os caminhos que existem hoje e não existirão depois.



Dois pontos de imposição estão por trás desse diagrama. A regra de rede nos hosts do banco de dados admite conexões apenas do gateway, das redes dos próprios serviços, do pipeline e do credential broker. E o gateway só encaminha uma sessão para o banco de dados que o tipo da sessão permite: uma sessão de leitura rotineira vai para `ledgerly_replica_2`, e uma sessão aberta para fazer uma requisição JIT ou break-glass vai para `ledgerly_primary`, como o Capítulo 6 estabeleceu.

Escreveria a regra de rede como uma lista curta que qualquer pessoa possa ler, porque a lista é o que uma revisão de acessos verifica. O Capítulo 6 definiu a diferença: esta lista é a *network policy*, que decide quais hosts podem se conectar, ao passo que a *connection policy* decide qual login pode se autenticar em qual banco de dados. Se ela contiver uma entrada que ninguém sabe explicar, essa entrada é um caminho.

```

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

```

A mesma lista continua com o que sai dos hosts do banco de dados e o que pode alcançar o 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

```

Observe o pouco que a VPN tem a fazer nesse modelo. Ela não protege mais o banco de dados. Ela leva as pessoas ao gateway, e o gateway decide o resto. A VPN importa menos; o acesso de cada pessoa agora está atrelado a uma sessão com nome.

Duas entradas da lista são fáceis de esquecer. A primeira é o egress. Um host de banco de dados que pode abrir conexões de saída para qualquer lugar pode ser usado para enviar dados para fora, e uma allowlist curta (replicação, o log de auditoria, backups) transforma isso em um evento vi-

sível. A segunda é a exposição do próprio gateway: ele agora é a única porta alcançável, então ele mesmo está em uma lista curta, e não pode ser alcançável de nenhum lugar que a VPN já não alcance.

A lista também supõe que a rede consegue distinguir quem chama. Em uma rede de cluster compartilhada, em que vários serviços vêm da mesma faixa de endereços, ela não consegue: uma regra que admite “as redes dos serviços” admite todos eles. Nesse caso, a regra de rede mantém fora tudo que não seja um serviço ou o gateway, e são o pooler e o login que separam um serviço do outro (capítulos 5 e 6). Registraria isso por escrito como a regra do cluster, para que ninguém leia a lista de rede como mais precisa do que ela é. Qual login pode se autenticar em qual banco de dados é a connection policy do Capítulo 6, e este capítulo apenas remete a ela.

O access gateway na Ledgerly

Veja o que o gateway faz, na ordem em que faz, e o que acontece quando um passo falha. Esta é a parte que testaria com cuidado, porque um gateway que falha aberto vira a nova senha compartilhada com uma marca melhor.

O que é uma sessão

Uma sessão começa quando uma pessoa se autentica. O gateway verifica a identidade, decide que tipo de sessão a pessoa tem direito de pedir (uma sessão de leitura rotineira para a maioria; os tipos JIT e break-glass apenas pelas requisições do Capítulo 8) e então cria um **login de sessão**, a conta que a sessão da pessoa usa. Ele é criado em `ledgerly_primary` e replicado (Capítulo 6). A sessão então se conecta ao banco de dados ao qual seu tipo pertence. Esse login carrega a identidade da pessoa no nome, de modo que aparece no log do banco de dados, e é membro de `human_readonly` por padrão. A credencial dele é gerada para essa sessão, entregue ao gateway e nunca mostrada à pessoa, então não há nada para salvar.

Um nome de login tem uma forma fixa, para que uma linha de log possa ser lida sem consulta: um prefixo que diz que é um login de sessão, o identificador curto da pessoa e o número da própria sessão, como em

`sess-robin-0412`. O nome identifica; é o registro do gateway que o associa à pessoa e à requisição, e nada deve interpretar o nome para decidir o que a sessão pode fazer.

Para criar e remover logins, e para conceder e revogar participações em papéis, o gateway precisa de privilégios no banco de dados. Ele os obtém por meio de seu próprio login para o papel `ledgerly_access_admin`. A etapa do pipeline para as definições de papéis e o credential broker também usam esse papel, cada um por meio de seu próprio login (capítulos 3 e 4). Isso faz do login do gateway uma credencial sensível: quem a detém pode conceder qualquer coisa dentro dos papéis que ela administra. Manteria essa credencial no mesmo lugar e sob as mesmas regras das credenciais do proprietário, não lhe daria nenhum privilégio sobre dados e auditoria cada uso dela (Capítulo 14). Copiá-la em scripts ou no chat por conveniência recriaria a senha compartilhada um nível acima.

Quem pode abrir qual tipo de sessão

`mayRequestKind` é política, e a manteria como dados que um revisor consiga ler, e não como uma condição enterrada no código do gateway. Uma tabela a responde: o grupo do single sign-on, os tipos que esse grupo pode abrir, a duração máxima de cada um, o banco de dados em que ele cai, seus padrões (somente leitura, statement timeout, row cap) e se pode fazer 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
```

Os tipos ligados ao primary são os sensíveis, já que uma sessão no primary está a uma requisição de distância do acesso de escrita. Quatro regras os mantêm raros: só grupos nomeados podem abrir um; ele precisa ser aberto com um motivo registrado em log; o gateway espera apenas um tempo

curto, digamos quinze minutos, para que uma requisição seja feita a partir dele; e uma sessão que termina sem uma requisição dispara um alerta, porque “abriu o primary e não fez nada” é um padrão que vale a pena notar. O limite de duas horas vale somente até um grant: quando `grantJitAccess` ou `declareBreakGlass`, do Capítulo 8, aprova uma requisição, o gateway redefine o `ends_at` dessa sessão para o `expires_at` da requisição, e `endSession`, abaixo, encerra a sessão nesse momento. O Capítulo 8 parte do outro lado, com a requisição.

A ordem das operações

O gateway, o log de auditoria, o banco de dados e o single sign-on são sistemas separados. Nenhuma transação abrange todos eles, então a ordem dos passos e o tratamento de uma sessão pela metade precisam ser projetados.

```

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))

```

Leia o bloco pelo tratamento de falhas. O registro é escrito antes de o login existir, então uma queda no meio deixa um registro que diz “opening” e nenhum login, ou um login que o registro explica. A entrada de auditoria vem antes do login: se o armazenamento do log estiver inalcançável, nenhuma sessão se abre. Essa escolha tem um custo, que a seção de trade-offs nomeia. Para uma sessão de leitura rotineira, o gateway espera até que `ledgerly_replica_2` tenha recebido o login, porque o replication lag faria, de outro modo, a primeira conexão falhar de vez em quando. Se a criação do login ou a concessão do papel falhar, o gateway tenta limpar e recusa a sessão. Se até a limpeza falhar, existe um login sem privilégios, e o sweep abaixo o remove.

Encerrando uma sessão

Uma sessão termina de uma de três maneiras: a pessoa se desconecta, o limite de ociosidade passa ou o `ends_at` passa. As duas últimas importam porque as pessoas deixam abas abertas. Uma conexão aberta sobrevive a uma credencial expirada (Capítulo 4), então o gateway fecha as conexões por conta própria e depois remove o login. A mesma função atende a uma pessoa que sai da empresa: um evento de desprovisionamento do single sign-on chama `endSession`, e a duração máxima da sessão é o limite para o caso em que esse evento nunca chega.

```
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 })
```

Aqui `auditOrQueue` acrescenta o registro ao log de auditoria e, se o log estiver inalcançável, o coloca em fila para novo envio e notifica a equipe de plataforma. A revogação e a remoção já aconteceram nesse ponto, então uma indisponibilidade do log atrasa a entrada e nunca a remoção do acesso. Um login que já não existe é tratado como sucesso: uma sessão que o

caminho de expiração do Capítulo 8 encerrou antes é fechada sem problemas aqui, seu registro é marcado como encerrado e nenhum alerta dispara.

O sweep chama `endSession` para cada sessão que já deveria ter acabado.

```
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")
```

A ordem é o ponto. Os grants de uma requisição são revogados a partir do registro armazenado e marcados como revogados somente depois que a revogação funcionou, e só então o login é removido, porque alguns motores se recusam a remover um login que ainda detém privilégios. Uma revogação ou remoção que falha notifica a equipe de plataforma e deixa o registro para o próximo sweep; uma escrita de auditoria que falha é colocada em fila e não interrompe o loop. O sweep dá um período de carência a uma sessão que ainda está “opening”, para não remover um login que tem apenas alguns segundos. A verificação de drift do Capítulo 3, que compara os logins que existem com os registros do gateway e os grants com as triplas armazenadas, roda em seu próprio agendamento, de modo que um job parado não esconda o outro.

Qual banco de dados, qual papel

Toda sessão cai onde seu tipo pertence. Uma sessão rotineira é um login `human_readonly` que se conecta a `ledgerly_replica_2`. Uma sessão aberta para fazer uma requisição JIT ou break-glass é aberta em `ledgerly_primary`, e o Capítulo 8 mostra como a requisição é feita a partir dessa sessão e como os grants terminam com ela. O gateway impõe a regra de destino que o Capítulo 6 escreveu como `connection policy`: é o gateway que conhece o tipo, e é o host do banco de dados que recusa um login destinado a outro.

Uma pessoa pode ter duas sessões abertas ao mesmo tempo, uma rotineira e uma com uma requisição. Manteria isso possível e manteria as sessões visivelmente separadas, para que ninguém use a privilegiada para uma consulta casual. O custo humano é uma segunda conexão no cliente da pessoa. O benefício é que a sessão privilegiada é fácil de encontrar, curta e fácil de encerrar.

O que o gateway registra

Como o gateway da Ledgerly fica no caminho dos dados, ele pode registrar cada instrução que uma sessão envia, junto com o início, o fim e as linhas afetadas. Guardaria esse registro onde as pessoas que ele descreve não possam editá-lo; o Capítulo 14 trata de onde. Um registro tem 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
```

Os regimes de dados de pagamento e de relatórios financeiros tendem a perguntar quem tocou os dados de produção e por qual caminho; o registro de sessão é essa resposta, e as pessoas de jurídico e de conformidade de quem lê são responsáveis pela interpretação. Dois limites vale conhecer. Um log de instruções diz o que foi perguntado, não o que foi devolvido, e não guardaria conjuntos de resultados completos, porque transformariam o log em uma cópia dos dados. O fingerprint carrega a forma da instrução e um hash de seus parâmetros em vez de seus valores, pelo mesmo motivo. E um log que ninguém lê registra sem proteger nada, então decida quem o examina e quando: uma amostra semanal de sessões que tiveram uma requisição JIT é um começo, e a revisão de acessos é a outra ocasião.

Capacidade e o orçamento de conexões

Toda sessão humana também é uma conexão de banco de dados, e o orçamento do Capítulo 5 deu às pessoas uma fatia fixa, digamos vinte conexões, como ilustração. As sessões rotineiras de engenharia caem todas em `ledgerly_replica_2`, de modo que essa réplica é o host da fatia humana, e o limite de conexões dela é onde o número finalmente é imposto. O gateway deveria impor essa fatia por conta própria: um limite por pessoa, um limite por tipo de sessão e uma recusa clara quando o orçamento se esgota. Uma recusa no gateway é uma mensagem para uma pessoa. Uma recusa no banco de dados, quando o orçamento acabou, pode expulsar um serviço. Pôr o limite no gateway impede que o orçamento humano vire problema do serviço.

O laptop e a cópia local

O gateway controla o caminho até o banco de dados. Ele não controla o que acontece com os dados depois que chegam a um laptop, e é aí que mora uma parte silenciosa do risco.

Duas coisas ficam no laptop. Uma é a configuração da conexão: o host salvo, o login salvo e, no estado inicial, a senha salva. A outra é o que a pessoa faz com um resultado: um arquivo exportado, uma tabela colada, uma cópia local de alguns milhares de linhas para “brincar”. Cada uma é uma cópia de dados no sentido do Capítulo 6, um lugar onde dados de produção vivem sem responsável e sem expiração.

Trataria isso em três camadas, da mais barata à mais cara.

Remover o que o cliente consegue lembrar. Como as credenciais de sessão são geradas para uma sessão e nunca mostradas, um cliente desktop não tem senha para salvar. A conexão salva agora guarda um endereço de gateway e o single sign-on da pessoa, e as entradas salvas antigas para o primary podem ser apagadas durante a migração, com uma lista curta do que apagar e onde. Essa camada quase não custa nada e remove o vazamento mais comum.

Escolher padrões que combinem com a tarefa. O gateway pode abrir sessões com padrões de somente leitura e um statement timeout do Capítulo 5. Ele também pode limitar as linhas que uma instrução devolve: con-

ta as linhas enquanto as transmite, fecha o cursor no limite, avisa a pessoa e grava o evento no registro de instruções; uma sessão que atinge o limite repetidamente é o que a amostra semanal examina primeiro.

Padrões que a pessoa pode suspender não continuam sendo padrões. Um cliente pode enviar um `SET` que aumenta o timeout ou desliga o somente leitura, então o gateway mantém uma lista de guarded settings e recusa qualquer instrução que os altere. O custo é que alguns clientes enviam tais instruções ao conectar e vão quebrar; é um custo a nomear na migração, e o Capítulo 9 trata de como o gateway decide quais instruções recusar. Clientes que fazem auto-commit são menos perigosos quando a sessão não tem privilégios de escrita com que fazer o commit. Um console no navegador, executado pelo próprio gateway, vai um passo além, porque nada é baixado a menos que exista um botão de export e que ele seja registrado em log. Esse console é mais lento e menos agradável do que um cliente nativo, e o ofereceria como opção para quem só olha e nunca precisa de ferramentas próprias, não como exigência.

Decidir explicitamente o que sai. Se um resultado pode ser exportado de uma sessão é uma regra à parte, e ela pertence ao trabalho de mascaramento e minimização do Capítulo 12 e aos caminhos de export do Capítulo 13. O gateway pode aplicar a regra, mas não pode torná-la desnecessária. Uma pessoa pode ler uma tela, fotografá-la ou redigitar um valor, e não conheço um controle com bom custo-benefício para isso. O gateway reduz a quantidade de dados que sai por acidente e torna visíveis os exports deliberados; ele não torna impossível sair.

Nota: Se seus engenheiros precisam de extrações grandes para análise, a resposta geralmente não é um limite de export maior no gateway. É uma cópia de analytics mascarada, projetada para esse fim, que os capítulos 6 e 12 descrevem. O gateway é a ferramenta errada para movimentação de dados em massa.

Uma porta, um ponto único de falha

Um ponto de entrada único é fácil de explicar, fácil de registrar em log e fácil de fechar. Ele também é um ponto único de falha, e vale dizer isso com clareza antes de recomendá-lo. Se o gateway estiver fora do ar, nin-

guém consegue alcançar o banco de dados pela porta da frente, inclusive quem está de on-call e precisa do banco de dados justamente quando outra coisa já quebrou.

Não há resposta que remova esse custo. Há respostas que o movem para um lugar que você consegue tratar.

Mantenha o gateway pequeno. Ele autentica, cria logins, encaminha e registra. Cada funcionalidade extra é mais um motivo para ele cair. Resistiria a acrescentar transformação de resultados e uma interface de aprovações ao mesmo serviço, e os executaria como componentes separados que o gateway chama, mas dos quais não depende para iniciar uma sessão de leitura. Onde uma instrução é verificada antes de rodar, essa verificação também é um componente à parte (Capítulo 9).

Execute mais de uma instância e coloque o estado em um lugar durável. Duas instâncias atrás do mesmo endereço, compartilhando os registros de sessão, significam que um restart ou um deploy ruim não encerra todas as sessões. O armazenamento de sessões e o log de auditoria também são dependências, e a disponibilidade deles faz parte da disponibilidade do gateway. É por isso que a função de abertura acima recusa uma sessão quando o log está inalcançável: é uma escolha deliberada que liga o acesso humano ao log, e deve ser uma escolha que a equipe fez de propósito e registrou por escrito.

Decida quais são os modos degradados. Há duas falhas para planejar. Uma é o gateway estar fora do ar enquanto os bancos de dados estão bem. A outra é o log de auditoria estar fora do ar enquanto o gateway funciona. Para cada uma, alguém responsável precisa ter decidido de antemão se o acesso humano para ou continua por uma rota mais estreita, e o que é registrado nesse meio-tempo. A decisão do livro para uma indisponibilidade do gateway é que não há uma segunda porta: o acesso humano para, break-glass inclusive, já que uma sessão break-glass é uma sessão do gateway com um papel mais pesado. A forma de manter isso raro é a redundância acima, e não um login de emergência fora do gateway. O Capítulo 8 dá o raciocínio e o que uma equipe faz se decidir de outro modo, e o Capítulo 15 descreve como ensaiar a indisponibilidade. Para uma indisponibilidade do log de auditoria, as leituras em uma réplica podem armazenar entradas em buffer e as escritas não (veja os trade-offs abaixo).

Trate o gateway como produção. Ele tem um responsável, um caminho de on-call, um health check que uma pessoa consegue ler e um processo de deploy com rollback. A equipe de plataforma é a responsável por ele. Se o gateway é o projeto paralelo de uma única pessoa, o ponto único de falha é essa pessoa.

O gateway também concentra um risco diferente: o comprometimento do gateway é o comprometimento do acesso a tudo que ele alcança. É por isso que seu login `ledgerly_access_admin` é protegido como descrito antes, e isso não é motivo para evitar o gateway: o estado inicial tem uma senha compartilhada, copiada para toda a equipe, na mesma posição.

A migração da Ledgerly de muitos caminhos para um

Uma equipe na posição da Ledgerly não trocaria tudo em um só dia. Seguiria os passos em uma ordem que mantenha um caminho funcionando disponível em cada ponto.

1. **Inventariar os caminhos.** Liste todas as formas pelas quais uma pessoa pode alcançar qualquer banco de dados da Ledgerly hoje: as rotas da VPN, o bastion, as entradas salvas nos clientes, quaisquer scripts que se conectem com o login de admin. A caminhada de Alex na abertura foi este passo, e ele deve terminar em uma lista escrita que a equipe de plataforma mantém.
2. **Subir o gateway ao lado dos caminhos antigos.** No início, ele cria logins de sessão no primary, onde os logins são definidos, concede a eles `human_readonly` e os conecta a `ledgerly_replica_2`, e nada além disso. Os caminhos antigos ainda funcionam. Algumas pessoas usam o gateway para a leitura diária e relatam o que é incômodo.
3. **Mover a leitura primeiro.** A leitura é a maior parte do que as pessoas fazem e é o que apresenta menos risco para experimentar. Depois que as partes incômodas forem corrigidas, feche a réplica para a VPN e apague as entradas salvas que apontavam para ela.
4. **Mover o primary por último.** O acesso direto ao primary é aquilo em que as pessoas se apoiam durante um incidente, então só o fecharia quando as rotas JIT e break-glass do Capítulo 8 estivessem funcionan-

do e ensaiadas. Fechá-lo antes tiraria o caminho de emergência antes de a substituição estar pronta.

5. **Remover o bastion antigo e o alcance do login de admin.** O bastion do prestador de serviço é desligado. O login de admin compartilhado deixa de ser aceito a partir de redes humanas. Os serviços mantêm seus próprios logins até que o trabalho do Capítulo 4 os substitua.
6. **Testar as negações.** Para cada caminho antigo, alguém o tenta e confirma que ele falha. Um caminho negado que ninguém testou é uma suposição.

O custo da migração é sobretudo humano. As pessoas precisam aprender um novo fluxo de conexão, e alguns de seus scripts vão quebrar porque supunham um login permanente. Planeje uma janela em que os dois funcionem e registre por escrito a data em que o caminho antigo fecha. Sem data, o caminho antigo continua aberto, porque sempre parece mais barato deixá-lo.

Trade-offs e custos

Um design menor que pode bastar. Um gateway é um sistema novo no caminho crítico, e se você não consegue manter pessoas para o caminho de on-call e para os deploys dele, um design menor é o melhor começo. Se quatro engenheiros alcançam um banco de dados por uma rede privada, usando logins pessoais de curta duração emitidos por algo que sua equipe já opera, você talvez já tenha a maior parte do valor. Registraria por escrito o que faria você mudar de ideia: a chegada de um prestador de serviço, um segundo banco de dados, uma requisição de gravações de sessão ou o primeiro incidente em que o log não soube dizer quem fez. O gateway é uma resposta a essas condições, não um marco.

Falhar fechado tem um preço. Recusar uma sessão quando o log de auditoria está inalcançável mantém o registro completo e faz do armazenamento do log uma dependência de toda sessão humana. Algumas equipes preferem abrir sessões de leitura com um buffer local e enviar as entradas depois. Essa é uma escolha defensável para sessões somente leitura em uma réplica, e mais fraca para sessões que escrevem. Se você a escolher, diga a quais sessões ela se aplica e monitore o buffer.

O registro de instruções é uma cópia de dados. O registro de fingerprints faz redaction dos literais, mas ainda nomeia tabelas e colunas, e qualquer campo deixado sem redaction, como uma nota de texto livre ou um comentário fornecido pelo cliente, pode carregar um valor. Trate o log com a classificação da coisa mais sensível que um registro possa conter, restrinja quem o lê e defina um período de retenção do qual alguém seja responsável.

Latência e fidelidade. Um gateway no caminho dos dados acrescenta um salto e pode não oferecer suporte a todo recurso de cliente nem a toda extensão de protocolo. Algumas ferramentas se comportam de modo estranho através de um intermediário: transferências muito grandes, cursores de longa duração e fluxos de autenticação incomuns são os lugares habituais. Teste as ferramentas que suas pessoas realmente usam antes de prometer que o caminho antigo vai fechar.

Quando não usar um gateway. Um protótipo de um único serviço sem dados de clientes não precisa de um. Um sistema cujo único acesso humano é por um console gerenciado que já oferece identidade, credenciais de curta duração e logs pode não precisar de uma segunda camada na frente. Um bastion é uma resposta razoável quando a necessidade real é um único lugar para fechar a porta da rede e quando o sistema protegido tem sua própria autenticação forte, por pessoa. Nos dois casos, valem as mesmas perguntas: quem é a pessoa, quanto dura o acesso e onde está o registro.

Erros comuns

- **Tratar a VPN como a camada de identidade.** É tentador porque a VPN já sabe quem está nela e todos a têm. Ela comprova a participação na rede, não qual login de banco de dados foi usado nem o que ele fez. Mantenha a VPN como o meio de alcançar o gateway e acrescente uma camada que associe uma pessoa a cada sessão.
- **Pôr um bastion na frente do banco de dados e dar o trabalho por encerrado.** Ele arruma a rede e deixa o login compartilhado e as instruções invisíveis. Escolha um bastion pelo que ele faz, uma única porta de rede que se pode fechar, e procure identidade e registro em outro lugar.

- **Deixar o caminho antigo aberto durante a migração.** Parece prudente manter os dois caminhos por um tempo, e é, mas sem uma data de encerramento o antigo sobrevive ao novo. Escreva a data, anuncie-a e teste que o caminho antigo falha depois.
- **Abrir uma sessão cujo fim ninguém impõe.** Um limite de sessão que existe só na documentação é um grant permanente com um nome mais longo. Encerre as sessões em `ends_at`, feche as conexões além de remover o login e execute um sweep que encontre sessões e logins que não deveriam existir.
- **Pular a decisão sobre o modo degradado.** Uma equipe que constrói um gateway sem decidir o que acontece quando ele está fora do ar decidirá sob pressão, em geral reabrindo o antigo caminho compartilhado “por um minuto”. Decida a rota de emergência de antemão, registre-a por escrito e ensaie.
- **Registrar tudo e não ler nada.** Capturar cada instrução parece diligência. Sem um responsável que faça amostragem e sem uma revisão de acessos que o examine, o log é uma cópia de dados sem finalidade. Decida quem o lê e o que essa pessoa procura.
- **Ler a lista de rede como mais precisa do que ela é.** Em uma rede de cluster compartilhada, a regra não distingue um serviço de outro, e um revisor que supõe que ela distingue aprovará um caminho mais largo do que parece. Registre por escrito qual controle separa quem chama.

Pontos-chave

- O alcance de rede e o acesso ao banco de dados são perguntas separadas, e um design que responde às duas com o mesmo controle não consegue dizer qual pessoa fez o quê.
- Um bastion torna o caminho de rede singular, mas não entende o protocolo do banco de dados, então, sozinho, deixa o login compartilhado e as instruções invisíveis.
- Um access gateway autentica uma pessoa por meio de single sign-on, cria um login de sessão que carrega essa identidade e o remove quando a sessão termina, e é isso que faz o log do banco de dados nomear uma pessoa.

- A regra de rede que permite apenas ao gateway alcançar os hosts do banco de dados é o controle que faz do gateway a única porta, e deve ser uma lista curta que uma revisão de acessos consiga ler.
- Um ponto de entrada único é um ponto único de falha, então o gateway deve ser pequeno, executar em mais de uma instância, pertencer à equipe de plataforma e vir acompanhado de uma decisão sobre o modo degradado tomada antes do incidente.
- O gateway não consegue controlar o que uma pessoa faz com dados que chegaram a um laptop; ele pode remover credenciais salvas, definir padrões de somente leitura e tornar os exports visíveis, e o resto pertence ao mascaramento e à minimização.
- Migre a leitura primeiro, feche o primary por último, defina uma data de encerramento para cada caminho antigo e teste que os caminhos negados estão negados.

O gateway dá a cada pessoa uma sessão com um nome e um padrão de leitura, mas a leitura raramente é todo o trabalho. O próximo capítulo examina o que acontece quando alguém precisa de mais do que `human_readonly` por um tempo limitado: o acesso just-in-time, o acesso break-glass e o que a aprovação custa a quem espera por ela.

Capítulo 8 — Acesso just-in-time e break-glass

Imagine uma noite de terça-feira na Ledgerly, pouco depois das três da manhã. Um alerta avisa que um lote de faturas está preso no status `sending` desde que o `invoice-worker` reiniciou, e os e-mails de lembrete da execução da manhã não sairão até que o lote seja liberado. Nada está exposto e nada se perdeu; o custo do problema é um atraso. Robin, quem está de on-call no backend, abre uma sessão de rotina pelo access gateway. A sessão é somente leitura e roda em uma réplica, que é o que Robin quer nos primeiros dez minutos: olhar, contar, entender.

A olhada confirma o problema, e a correção exige um `UPDATE` em uma coluna de `invoices`. Robin escreve a correção de uma única instrução e abre uma requisição para ela. Quem aprova é a pessoa de on-call secundária, que está dormindo, e o page ainda não recebeu ack. Robin tem um laptop, um runbook e uma decisão a tomar: esperar uma resposta ou achar um jeito de contornar a pergunta. Na Ledgerly antiga, essa decisão nem existiria, porque a senha de administrador compartilhada estava a um copiar e colar de distância. Na Ledgerly nova, o caminho de contorno precisa ser projetado, e o caminho de volta também.

Este capítulo acrescenta os próximos passos do estado-alvo do exemplo corrente: as pessoas não têm acesso permanente de escrita, as escritas chegam como grants just-in-time dentro do teto do Capítulo 3, e existe um papel de emergência que é raramente usado e sempre revisado. Nas cinco partes do livro, trata-se sobretudo do Grant: quem aprova o quê, para qual instrução, por quanto tempo. O capítulo responde à pergunta de Robin: como o acesso aparece quando é necessário, como ele termina junto com a tarefa e o que acontece quando o caminho normal é lento demais?

O problema e por que ele é difícil

O acesso permanente é a promessa de que ninguém precisará pedir de novo. É conveniente por um mês e, depois disso, ninguém se lembra do que a promessa cobria. As pessoas mudam de equipe e mantêm seus grants; um papel ampliado para um incidente continua amplo. Quando chega uma revisão de acessos (Capítulo 14), quem revisa se depara com grants sem nenhuma tarefa associada, e a resposta para “isso ainda é necessário?” é “provavelmente”. O custo de um grant permanente é cada dia em que ele não é usado, porque ele continua carregando o raio de impacto de tudo o que cobre.

A substituição óbvia é o **acesso just-in-time (JIT)**: acesso concedido para uma tarefa específica e por um tempo limitado. É fácil de enunciar e fácil de errar de forma sutil em cinco pontos.

Escopo. A implementação fácil concede um papel a um login por uma hora. Se esse papel for `human_jit_write`, o login recebe o teto inteiro, direitos de update em seis tabelas para uma tarefa que precisava de uma coluna de uma só, e o grant é um grant permanente com um cronômetro preso a ele. O escopo precisa estar nos próprios grants, uma requisição por vez.

Expiração. Nos motores comuns, os grants não expiram sozinhos. Algo precisa retirá-los, em geral um job agendado, e um job pode parar, perder sua credencial, falhar em um login e seguir adiante, ou rodar contra a lista errada.

Aprovação. Uma segunda pessoa lendo cada requisição tem um custo humano: alguém é acordado, alguém espera. Se as aprovações são lentas demais, as pessoas encontram o caminho mais rápido, que é a senha compartilhada que o livro tenta aposentar. Se o aprovador sempre responde em segundos, a aprovação deixa de reduzir a exposição.

A emergência. Um esquema de acesso sem um jeito sancionado de agir depressa faz crescer um jeito não sancionado. A questão é se o caminho rápido foi projetado, registrado e revisado, ou improvisado.

A operação. Um grant nomeia uma tabela e uma coluna e não diz nada sobre quais linhas ou qual instrução. Um aprovador que lê “`UPDATE` em `invoices`, `status`” aprovou uma capacidade, e o que quer que rode den-

tro dela fica fora da aprovação. Para que a aprovação signifique “uma segunda pessoa viu o que vai acontecer”, a requisição precisa nomear a operação e o grant precisa decorrer dela.

Uma política que diz “o acesso deve ser temporário” não diz ao sistema o que fazer quando o job de revogação está fora do ar. O restante do capítulo liga grants, registros e cronômetros para que ele saiba.

Acesso just-in-time: requisições que nomeiam uma operação

Uma requisição JIT é construída com cinco elementos, cada um presente por causa de uma das falhas acima. Cada um vem marcado com a parte do modelo do livro a que serve.

- O teto (Grant), `human_jit_write`, define o máximo que uma requisição pode pedir. Ele nunca é concedido a um login (Capítulo 3).
- Uma requisição de escrita (Grant) nomeia a correção revisada a que se destina, em `fix_id`. Suas triplas (privilégio, tabela, colunas) derivam dessa correção, dentro do teto, e são armazenadas em `access_request_privilege`.
- A aprovação (Grant) é da instrução, e o grant impõe seus limites: o aprovador lê a instrução, o gateway concede exatamente as triplas derivadas ao login de sessão do solicitante, e o conjunto de instruções aprovado é registrado como `approved_hash`. A expiração revoga exatamente essas triplas, lidas do mesmo registro.
- As escritas em uma sessão JIT (Execução) são encaminhadas apenas pelo runner, que recusa qualquer coisa que não corresponda a `approved_hash`.
- Uma reconciliação independente (Registro) compara os grants que o banco de dados mantém com as requisições que os justificam, de modo que um job parado seja notado por algo que não seja o próprio job.

Vou tratá-los na ordem em que uma requisição os encontra.

O que uma requisição contém

Uma requisição é feita a partir da sessão que vai usá-la, e essa sessão roda no primary. Uma sessão carrega no máximo uma requisição ativa; uma tarefa que ultrapassa a sua janela continua em uma nova sessão com uma nova requisição. A investigação de rotina fica na réplica com `human_readonly` (Capítulo 6), e o Capítulo 7 diz quem pode abrir uma sessão ligada ao primary. O gateway sabe qual login pertence a qual pessoa, então ninguém digita um principal ou um login.

Eis o formato da requisição de Robin, antes que alguém a tenha 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")
```

Três coisas merecem atenção. O motivo aponta para algo verificável, um alerta ou um ticket. O grant é estreito, e Robin não precisa de nenhuma tripla `SELECT` porque a sessão já tem `human_readonly`. E Robin não digitou a tripla: ela foi derivada de `fix-0093`, de modo que o grant e a instrução que o aprovador lê não podem divergir. Uma requisição que contém apenas triplas `SELECT`, por exemplo para ler uma coluna Confidential sem mascaramento, não tem correção nem hash.

Um grant restringe privilégios e colunas, não linhas. O `UPDATE` acima pode alterar `status` em todas as faturas que a sessão consegue ver. Um grant JIT limita o alcance de um erro por coluna e por tempo; o que o limita por linha é a instrução que o aprovador leu e o runner executa, com a disciplina de contar primeiro do Capítulo 10.

Vinculando a requisição à instrução

A correção é um conjunto de instruções revisado, com seu filtro e sua contagem de linhas esperada, na forma que o Capítulo 10 descreve, e a requisição a nomeia por `fix_id` e carrega seu `approved_hash`. O hash vem do registro de merge da correção ou do registro da requisição aprovada, e nunca da entrada do runner, e em uma sessão de escrita JIT o gateway encaminha escritas apenas do runner. Uma requisição que contém apenas triplas `SELECT` não tem correção, hash nem runner, e o aprovador lê a lista de colunas. O Capítulo 9 enuncia a regra por completo e trata de como o gateway verifica cada instrução.

Verificando uma requisição

O gateway verifica o formato da requisição antes que alguém seja chamado a aprovar, e a verifica de novo, com o aprovador, no momento do 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

```

As linhas de autoaprovação são o ponto em que o four-eyes é imposto em código: quem pede, ou quem escreveu a correção, não pode ser quem responde. Uma requisição pendente não tem aprovador, e é por isso que a verificação de formato deixa essas linhas de fora; o aprovador é preenchido no momento da aprovação, e a verificação completa roda no momento do grant.

Recusaria uma janela longa demais em vez de encurtá-la em silêncio: uma janela encurtada deixa o solicitante achando que tem uma hora quando tem vinte minutos, e a surpresa chega no meio do reparo.

`withinCeiling` é onde o teto do Capítulo 3 se torna exigível. Ele rejeita uma tripla cuja tabela esteja fora do teto, cujo privilégio não seja oferecido naquela tabela, ou cuja lista de colunas inclua `iban`, `account_holder_name` ou `token_value`. Rejeita `INSERT` e `DELETE` em `bank_accounts` e `payment_tokens`, porque uma linha nova ou removida define ou destrói valores Restricted. Rejeita qualquer tripla, exceto `DELETE`, que não tenha lista de colunas, salvo uma tripla em nível de tabela em um motor que use o fallback das tabelas de secrets. E rejeita qualquer coisa que toque `ledger_entries`, `audit_events` ou os registros de requisições de acesso, nos quais nenhum papel humano escreve.

Uma coisa fica dentro do teto de propósito. `UPDATE (customer_id)` em `bank_accounts` ou `payment_tokens` passa por `withinCeiling`, porque `customer_id` apenas liga uma linha ao seu proprietário e não é `Restricted`. Mover a linha para outro cliente ainda muda de quem é a conta ou o token, então a política de execução o encaminha ao caminho de produto (Capítulo 9). O checklist do aprovador pega o caso manualmente.

Quem aprova, por quanto tempo, para qual escopo

Uma requisição JIT tem três variáveis: o aprovador, a janela e o escopo.

O aprovador é uma pessoa que entende as tabelas da requisição e que não é nem quem solicitou nem quem escreveu a correção. Na Ledgerly, o aprovador natural para `invoices` é outra pessoa de backend, em geral a de on-call secundária. O aprovador natural para qualquer coisa em `payments` ou `payment_tokens` é alguém da equipe responsável pelo `payments-service`, porque essa equipe sabe o que um `status` alterado ali vai disparar adiante. A equipe de plataforma opera as ferramentas de acesso, e não a tornaria o aprovador padrão para dados que não são dela: ela vira o gargalo, e um aprovador que não consegue julgar a mudança.

```
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
```

Um pool vazio falha de forma visível e nunca recorre a um mais amplo. Fora isso, a ordem do pool é a ordem de escalonamento: o page vai para a primeira pessoa, depois para a seguinte, dentro da mesma requisição pendente, que só é recusada quando chega a `expires_at` sem resposta.

A janela deve ter a duração da tarefa típica, não a da pior tarefa. Digamos que uma correção de dados costume levar vinte minutos, o formulário ofereça uma hora e o máximo do gateway seja de quatro horas. Estender um grant é uma nova requisição com uma nova aprovação, não a edição da

antiga, porque uma edição deixaria uma única aprovação se esticar indefinidamente. Quando um reparo ultrapassa a janela, a sessão termina e qualquer transação aberta sofre rollback junto, de modo que um reparo escrito como uma única transação (Capítulo 10) perde seu trabalho de forma limpa, em vez de deixar metade dele aplicada.

O **escopo** são as triplas e a instrução de que elas vêm. O aprovador lê ambas, e a tela de aprovação mostra as triplas como os grants em que elas vão se transformar: “UPDATE em `invoices (status)`, até 04:12”. Uma requisição que nomeia uma coluna Confidential em um `SELECT` mostra essa coluna sem mascaramento, e a tela diz isso em palavras, não apenas na lista de colunas.

A aprovação acontece na tela do próprio gateway, por single sign-on com uma verificação multifator nova no momento do clique, e não como reação sob uma mensagem de chat: a aprovação é o controle, então o que quer que a registre precisa saber quem a deu.

O que o aprovador verifica

A decisão do aprovador é mais estreita do que “isso é uma boa ideia?”: aprovar a instrução e deixar o grant impor seus limites. Este checklist espelha o checklist de quem revisa a correção no Capítulo 10, do lado do acesso.

1. O motivo é um alerta ou ticket verificável, e ele ainda está aberto?
2. A instrução na tela faz o que o motivo diz? Leia a instrução, não o resumo que o solicitante fez dela.
3. O filtro limita as linhas, e a contagem esperada confere com o que o solicitante viu?
4. As triplas derivadas cobrem a instrução e nada além? Sob o fallback das tabelas de secrets, “nada além” significa “todas as colunas não Restricted da tabela”, e a tela diz isso.
5. A janela tem aproximadamente a duração desta tarefa, e está abaixo do máximo?
6. A instrução toca `customer_id` em `bank_accounts` ou `payment_tokens`, ou qualquer coluna Restricted? Então não é uma correção de dados JIT. Envie-a ao caminho de produto e recuse a requisição.

7. Não sou nem o solicitante nem quem escreveu a correção, e me sentiria à vontade para explicar esta aprovação a quem revisar na semana que vem?

Um aprovador que não consegue responder a uma dessas perguntas pergunta ao solicitante enquanto a requisição permanece pendente. Silêncio não é aprovação, e uma lista de colunas também não.

Concedendo

A aprovação é o momento em que quatro registros precisam concordar: o registro da requisição no gateway, a entrada do log de auditoria, os grants no banco de dados e a sessão. Nenhuma transação os abrange, então a ordem importa, e importa também o que cada falha deixa para trás.

```

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)

```

Leia-o pelos seus modos de falha. Se o log de auditoria estiver fora do ar, a requisição é recusada e nada é concedido: um grant que não pode ser registrado não pode ser revisado, e prefiro interromper um reparo a criar um invisível. Se o gateway morrer depois do registro e da entrada de log, mas antes dos grants, a requisição fica aprovada e sem efeito, o que não expõe nada. Se morrer depois de duas das três triplas, o login mantém duas, o registro mantém três, e a revogação posterior retira as três, já que revogar uma tripla que nunca foi concedida é inofensivo.

As últimas linhas tratam de uma `race condition`. A janela não recomeça na aprovação, então a verificação de formato recusa uma aprovação que chegue depois de `expires_at`; o que resta é uma aprovação que passa um instante antes de a janela fechar, e essas linhas retiram os grants dela imediatamente.

A entrada de auditoria carrega a correção e o hash junto com as triplas, de modo que o aprovador, a instrução e a janela ficam em um único registro. Os regimes de dados de pagamento e de relatórios financeiros tendem a perguntar quem aprovou uma mudança em dados de produção, com que base e por quanto tempo, e este registro responde às três perguntas; a interpretação cabe ao jurídico e à conformidade.

Encerrando o grant

Um grant termina em três camadas, e nenhuma delas é tratada como a única.

A primeira camada é a sessão. O gateway a encerra em `expires_at` e remove o login, sem nenhum job agendado: o relógio que limita o acesso pertence ao componente que o criou. Uma sessão ligada ao primary abre com o teto de duas horas do Capítulo 7, então, em um grant, a partir de `grantJitAccess` ou `declareBreakGlass`, o gateway redefine o `ends_at` da sessão para o `expires_at` da requisição, e o `endSession` do Capítulo 7 encerra a sessão nesse momento; uma janela mais longa que duas horas, portanto, não é cortada.

```
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")
```

O encerramento em si é o `endSession`, o único lugar que fecha conexões, revoga o que as requisições da sessão concederam e remove o login, e o `sweepSessions` chama a mesma função (Capítulo 7).

A segunda camada é o job de revogação, que lê o mesmo registro e remove o que a requisição concedeu. O `revokeRequestAccess` deriva seu trabalho de `access_request_privilege`: para uma requisição JIT, ele revoga

cada tripla armazenada; para uma requisição break-glass, revoga o papel `human_breakglass`. Ele não lê uma lista separada que poderia divergir, e trata um login ausente como já revogado.

```
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)
```

A revogação continua quando o log de auditoria está fora do ar. A assimetria é deliberada: um grant sem entrada de log é uma exposição, enquanto uma revogação sem uma é uma lacuna que pode ser preenchida depois.

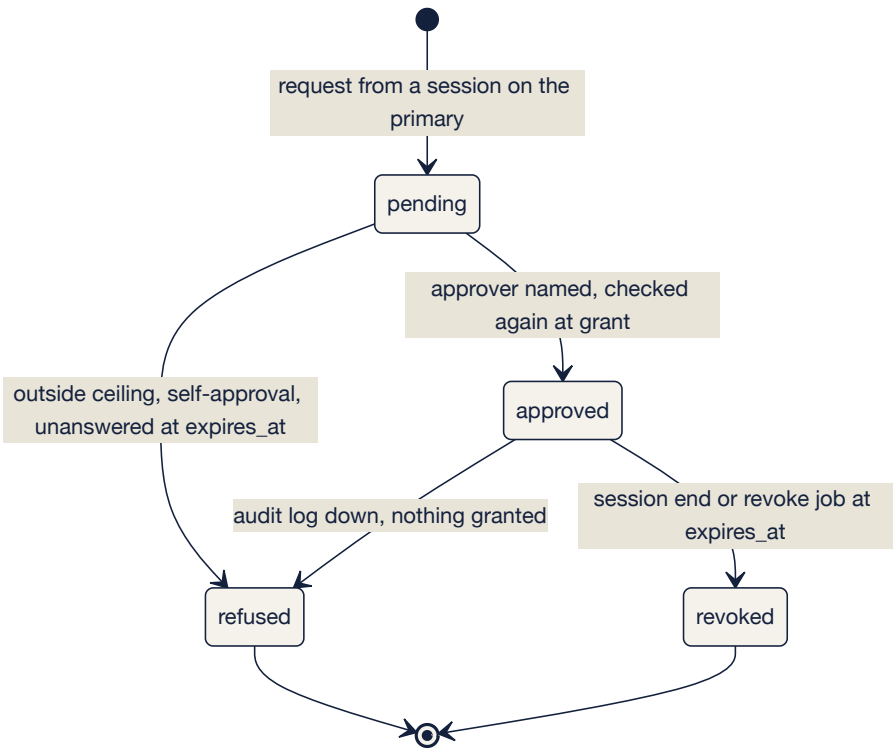
A terceira camada é a reconciliação, e é uma coisa de outra natureza. Ela não retira acesso. Ela pergunta, de fora, se as duas primeiras camadas fizeram o seu trabalho:

```
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")
```

Ao lado dela, o drift check do Capítulo 3 lê os grants que o banco de dados de fato mantém e os compara com as triplas não expiradas das requisições. Um grant direto sem requisição ativa por trás, um login que é membro do teto, um membro de `human_breakglass` sem requisição `break-glass` aberta: cada um é sinalizado, mesmo quando o job de revogação informa sucesso. A verificação de atrasos observa os registros sobre os quais o job deveria ter agido; o drift check observa o banco de dados que o job deveria ter alterado. Um job que informa “concluído” sem alterar nada só é pego pelo segundo.

A reconciliação não repara nada por conta própria, e manteria assim: uma revogação automática de grants desconhecidos é um jeito de quebrar a produção à noite. Ela envia uma notificação à equipe de plataforma, e uma pessoa decide.

O estado por que uma requisição passa é pequeno, e desenhá-lo mostra onde cada camada atua. Observe as saídas de `approved`: uma revogação, ou uma recusa quando o log de auditoria está fora do ar e nada foi concedido; nada volta de `revoked`.



Autoaprovação e nenhuma aprovação automática para on-call

A autoaprovação é sempre recusada. `checkJitRequest` levanta `SelfApprovalNotAllowed` e nada a sobrepõe, nem senioridade nem urgência. O caso urgente tem outro caminho, que é a próxima metade deste capítulo.

A aprovação automática é o outro atalho tentador, porque o atrito é maior à noite. Para leituras, não a construiria: todo login de sessão já tem `human_readonly`, e a única coisa que um `SELECT JIT` acrescenta é uma coluna Confidential exibida sem mascaramento, que é exatamente o que um aprovador deve examinar. Uma escrita muda aquilo sobre o que os serviços agem em seguida, e uma máquina não sabe dizer se `status` é a coluna certa. Mantenha as escritas com um aprovador humano e gaste o esforço em tornar esse aprovador fácil de alcançar.

Se você adicionar uma aprovação automática mais tarde, para uma leitura que de fato traga algo novo, a regra precisa exigir uma lista explícita de colunas, e uma lista vazia ou ausente deve recusar a requisição de forma visível. Sob o fallback das tabelas de secrets, uma leitura em nível de tabela não tem lista de colunas alguma, então uma regra que trate “nenhuma coluna Confidential nomeada” como aceitável aprovaria todas as colunas Confidential da tabela.

Requisições de automação e a política como código

Um job, um script de runbook ou um agente de IA que queira escrever é um principal como qualquer outro: abre sua sessão sob a própria workload identity, nomeia uma correção e espera por um aprovador humano, e o registro nomeia tanto a automação quanto a pessoa que a acionou (o Capítulo 9 trata do que ela pode fazer dentro de uma sessão). A lógica de aprovação em si é código. `withinCeiling`, `chooseApproverPool`, `maxJitWindow` e as verificações de formato ficam no repositório, passam por revisão e fazem parte da baseline do drift check; um teto editado à mão em um console é o problema original um nível acima.

Casos de abuso

Três casos são limitados pelos controles acima e eliminados por nenhum deles. O conluio entre solicitante e aprovador fica restrito às colunas, à instrução e à janela do registro, e a dupla fica visível na entrada de audito-

ria; a revisão de acessos (Capítulo 14) é onde aparece uma dupla que se aprova mutuamente com frequência muito acima do acaso. Um aprovador alvo de engenharia social (“aprove a req-0412 agora, o lote está bloqueando a folha de pagamento”) é enfrentado pelo checklist: leia a instrução na tela do gateway, verifique o alerta no motivo e não responda nada enviado por outro canal. Um token de single sign-on sequestrado não consegue escrever em uma sessão JIT, porque o gateway encaminha escritas apenas do runner e o runner recusa um hash que nunca foi aprovado. O risco residual está nas leituras dentro das triplas concedidas e no break-glass, que não tem aprovador; a verificação step-up na declaração e o alerta são o que o limitam.

Acesso break-glass

Break-glass é um acesso de emergência com um conjunto de logging definido e revisão obrigatória. O nome vem do interruptor de incêndio com frente de vidro: ao alcance de quem precisar dele, e quebrá-lo é visível para todos.

No modelo do livro, é o papel `human_breakglass`, e é o único papel humano que lê colunas `Restricted`. Ele lê todas as tabelas e escreve `INSERT` e `UPDATE` em todas elas, exceto nas duas somente de acréscimo; em `bank_accounts` e `payment_tokens` ele escreve apenas `UPDATE`, e apenas em colunas não `Restricted`. Ele nunca escreve em `ledger_entries` nem em `audit_events`, porque nenhum papel humano escreve. Não é um caminho para correções de rotina e não é um caminho de correção de dados: uma correção de dados é planejada, revisada e limitada (Capítulo 10), e o break-glass não é nada disso, por projeto.

Uma requisição break-glass é autodeclarada. Não tem aprovador. A pessoa na sessão afirma que existe uma emergência, dá um motivo, e o gateway marca a requisição como aprovada e concede o papel ao mesmo login de sessão até `expires_at`. O mesmo job de revogação e o mesmo cronômetro de sessão o retiram.

O aprovador é o controle inteiro do JIT, então um papel de emergência sem aprovador pode parecer estranho. O break-glass existe para os casos em que o pool de aprovadores não respondeu, e uma etapa de aprovação

reproduziria essa falha. O controle passa de antes do acesso para o redor dele: um alerta na declaração, o conjunto de logging abaixo, uma janela curta e fixa, e uma revisão que vem em seguida sem exceção.

```
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)
```

Três decisões nesse bloco precisam de um comentário.

A janela é fixada pelo gateway, em duas horas na ilustração da Ledgerly. Nenhuma janela mais longa está disponível mediante requisição, porque “preciso de um pouco mais” transforma uma exceção em hábito. Se a emergência ultrapassar a janela, a pessoa declara de novo e a segunda declaração alerta de novo; esse é o atrito pretendido.

A verificação do log de auditoria é a mesma do JIT. Se o log estiver fora do ar, a declaração é recusada. Aceito esse custo, porque uma sessão break-glass sem registro é a única coisa que uma revisão não consegue recons-

truir, e a revisão é o que torna o papel tolerável. O que o on-call faz quando o caminho normal e o log estão ambos fora do ar pertence ao Capítulo 15.

O alerta é uma notificação aos contatos de segurança e à equipe de plataforma de que alguém abriu o acesso mais amplo que a Ledgerly tem, não uma requisição de permissão. Em um sistema saudável, ele é raro o bastante para ser lido; se disparar toda semana, deixará de ser lido, e o break-glass estará sendo usado para algo que merece um caminho projetado.

O conjunto de logging do break-glass

“Logging pesado” não é um padrão até ser definido, então, para uma sessão break-glass, o conjunto de logging são estas quatro coisas:

- Cada instrução, como um fingerprint com seus literais sob redaction e um hash dos parâmetros, com o início, o fim e as linhas afetadas. A redaction importa mais aqui: esta sessão lê colunas Restricted, e um literal no log as copiaria para um lugar sem proteção em nível de coluna.
- As entradas à medida que acontecem. Cada instrução é acrescentada ao log de auditoria enquanto roda, não coletada no final, de modo que uma sessão interrompida ainda deixe o seu registro.
- Uma nota de uma linha em toda escrita. O gateway recusa uma escrita sem ela, o que custa segundos e dá a quem revisa a intenção ao lado da instrução.
- O mesmo registro de qualquer outra sessão: o login, a pessoa, a requisição e o motivo, para que os três registros do Capítulo 14 possam ser unidos.

Um risco permanece, e ele difere conforme o estado. Hoje, sem gateway ou com um que apenas intermedeia a credencial, uma sessão break-glass é o único lugar em que um `UPDATE` sem limite em uma tabela em que ela pode escrever ainda é possível. A janela, as contagens de linhas, as entradas e a revisão limitam o dano e o expõem; não o impedem. No estado-alvo, o gateway do Capítulo 9 executa sessões de escrita com auto-commit desligado e um row cap, sua matriz nega uma escrita sem limite neste lane, e uma escrita sem nota é recusada. O que continua possível é uma escrita que cabe dentro do cap e ainda assim atinge as linhas erradas. O

cap é mais difícil de dimensionar aqui, porque uma emergência, por definição, não é a tarefa planejada; até que a política defina um, a escrita sem limite é um risco aceito, e é a razão de a revisão existir.

Revisão posterior

Todo uso de break-glass é revisado, por alguém que não seja a pessoa que o usou. A revisão não é uma audiência; ela faz as perguntas que uma revisão de design faz: era uma emergência que o caminho normal não conseguia atender; o que a sessão leu ou escreveu; ela leu colunas Restricted, e isso era necessário; e uma requisição JIT poderia ter atendido à mesma necessidade, e, se sim, o que teria tornado essa requisição rápida o bastante?

A última pergunta é a que compensa. Cada revisão ou confirma que o break-glass foi a ferramenta certa ou encontra uma lacuna no caminho normal que o tornou a única ferramenta. Corrigir a lacuna (um aprovador ausente, uma janela curta, uma tabela que o teto não oferece) reduz o próximo uso. A revisão também precisa do seu próprio relógio:

```
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)
```

Uma revisão que ninguém faz é um controle que deixou de existir em silêncio, então o seu prazo dispara um alerta.

Quando o gateway não está disponível

As pessoas chegam ao banco de dados apenas pelo access gateway (Capítulo 7), e tanto os grants JIT quanto os break-glass são feitos pelo gateway a um login de sessão que ele criou. Uma indisponibilidade do gateway é, portanto, também uma indisponibilidade do break-glass. Isso é um trade-off do design, e o Capítulo 7 enuncia a mesma decisão do lado do gateway.

A alternativa é uma segunda porta: um login de emergência fora do gateway. Ela é atraente na noite em que é necessária e perigosa em todas as outras, porque uma credencial que funciona sem o gateway funciona sem o seu logging, a sua identidade e a sua expiração; é a senha de administrador compartilhada com um nome melhor.

Prefiro tornar o gateway difícil de perder a tornar a sua ausência suportável enfraquecendo todo o resto: mais de uma instância, poucas dependências (o log de auditoria, os registros de requisições), uma falha ensaiada e um sinal de saúde que o on-call veja primeiro. Se todas as instâncias caírem ao mesmo tempo, não resta nenhum caminho humano de dados, e não inventaria um aqui. O instrumento selado do Capítulo 3 existe para trabalho em nível de motor, precisa de uma rota de rede que o Capítulo 7 fecha para as pessoas e contorna as proteções de somente acréscimo e de dados Restricted, então não é um caminho de acesso a dados. O Capítulo 15 volta ao que fazer nesse caso.

A escolha não é gratuita. Uma equipe com um gateway frágil pode razoavelmente querer um segundo caminho enquanto ele amadurece. Nesse caso, escreva o que ele é, quem pode obter a sua credencial, como o seu uso alerta e o que o aposenta: um caminho com responsável e data de término é uma decisão, e um sem eles é um resquício.

Quando o motor não tem grants em nível de coluna

Tudo acima pressupõe que o motor consiga conceder um privilégio em colunas nomeadas. O Capítulo 3 apresentou o único fallback que o livro usa para motores que não conseguem: as colunas Restricted vivem em tabelas separadas, `bank_account_secrets` e `payment_token_values`, fora do teto e alcançáveis apenas por `app_payments_service` e `human_breakglass`. As tabelas principais então não contêm nenhuma coluna Restricted, de modo que um grant JIT em nível de tabela em `bank_accounts` ou `payment_tokens` não consegue alcançar nenhuma. Uma requisição assim não tem lista de colunas: `columns` é nulo, e a tripla diz “UPDATE em `payment_tokens`”.

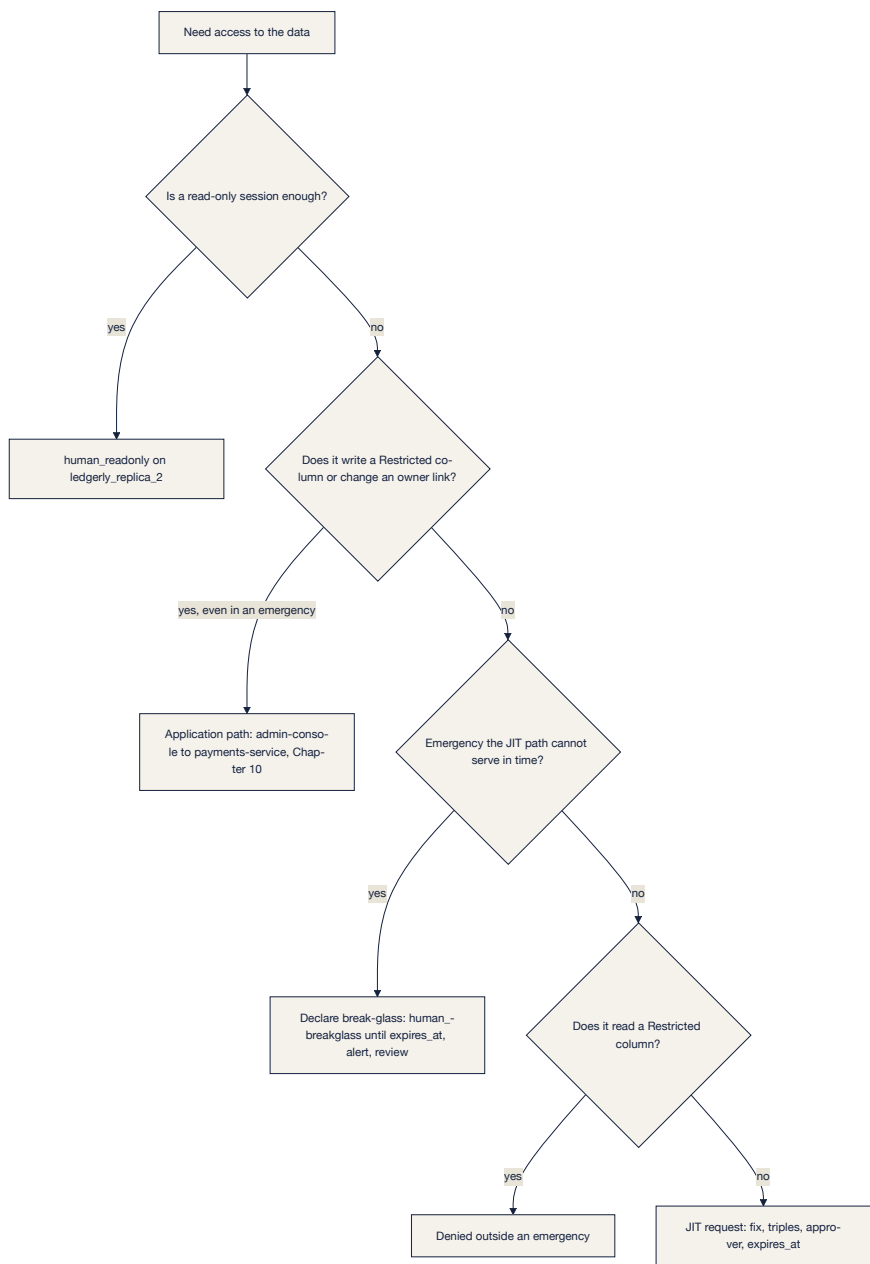
O preço recai sobre o aprovador, que aprovou todas as colunas não Restricted da tabela, inclusive as Confidential. Se o solicitante queria alterar `expires_on`, o aprovador também está aprovando `provider` e

`last_four`. A tela de aprovação precisa dizer “todas as colunas não Restricted desta tabela” em palavras, e a instrução que o aprovador lê passa a carregar mais peso. Um grupo de colunas que uma equipe espera abrir com frequência merece a sua própria tabelinha.

O fallback não muda as regras para dados Restricted. Uma pessoa continua sem nunca escrevê-los com SQL cru; a mudança passa pelo `admin-console` e pelo `payments-service` depois que uma segunda pessoa aprova (Capítulo 10).

Escolhendo o caminho

Uma pessoa às três da manhã precisa saber qual porta usar, e a resposta pertence ao runbook como um fluxo curto. O diagrama abaixo é o formato dele. Observe as duas pontas que nenhuma rota humana de dados atende: uma escrita em uma coluna Restricted, que vai para o caminho de produto mesmo em uma emergência, e a leitura de uma delas fora de uma emergência, que é negada. O design diz isso, em vez de deixar a decisão a quem está sob pressão.



A Ledgerly às 3 da manhã

De volta a Robin. As linhas presas foram contadas na sessão de rotina, e o número bate com o runbook. A escrita é a requisição vinculada a `fix-0093`, por uma hora, citando o alerta aberto. `chooseApproverPool` seleci-

ona a pessoa de on-call secundária e a liderança da equipe, nessa ordem, e o page vai para a primeira. A Ledgerly antiga teria aberto a senha compartilhada aqui. A nova tem uma espera, e a espera é o design funcionando.

Se ninguém responder, o runbook faz a pergunta que decide tudo: é uma emergência que o pool de aprovadores não consegue atender a tempo? Os lembretes presos podem esperar até de manhã, então Robin deixa uma nota no alerta e para, um desfecho que a senha antiga nunca permitiu.

Suponha, em vez disso, que o `payments-service` estivesse falhando, a cobrança de pagamentos tivesse parado para todos os clientes e o diagnóstico exigisse uma coluna `Restricted` que nenhuma requisição JIT pode conter. Esse é um caso para o `break-glass`. Robin declara com o alerta como motivo; o gateway concede `human_breakglass` pela sua janela fixa; os contatos de segurança e a equipe de plataforma são avisados. Na manhã seguinte, alguém que não é Robin lê o log e faz as perguntas da revisão, uma das quais é por que o diagnóstico precisou de uma coluna `Restricted`. A resposta pode levar a uma view de diagnóstico mais segura no próximo trimestre.

Trade-offs e custos

O atrito é o preço, e precisa ser gasto onde compra alguma coisa.

Uma requisição JIT custa ao solicitante um minuto para escrever e ao aprovador um minuto para ler, no melhor caso, e uma pessoa acordada no pior. Para uma tarefa que acontece duas vezes por trimestre, isso é barato. Para uma tarefa que acontece duas vezes por dia, é uma funcionalidade de produto com disfarce operacional, e o Capítulo 10 descreve como movê-la para o `admin-console`.

Vincular a uma correção custa preparação. A correção precisa ser escrita antes que a requisição possa ser aberta, o que é mais lento do que um grant sozinho. É o preço de aprovar o que vai rodar em vez do que poderia rodar.

A fadiga de aprovação é uma falha do design, não do aprovador. Uma aprovação que chega em segundos, sempre, da mesma pessoa é teatro: ou as requisições são triviais e `human_readonly` deveria cobri-las, ou o aprovador parou de ler. O remédio são requisições menores, uma tela que mostre o que os grants serão, aprovadores que conheçam as tabelas e um pool

grande o bastante para que ninguém seja o padrão. Dois sinais pertencem à revisão de acessos: com que frequência as requisições são recusadas ou devolvidas, e com que frequência o mesmo aprovador responde. Uma taxa de recusa zero tem tanta chance de significar que ninguém está olhando quanto de significar que as requisições são boas.

O on-call arca com o custo primeiro. O aprovador é acordado, o solicitante espera, e a equipe aprende um runbook que raramente usa. Faça do pool de aprovadores um rodízio, publique-o e garanta que as pessoas nele tenham o contexto para ler uma requisição sobre **invoices** ou **payments**.

O gateway é uma dependência de toda escrita. O job de revogação, o log de auditoria e os registros de requisições são todas dependências de um grant; quando um deles está fora do ar, os grants param, o que limita a exposição e atrasa a resposta a incidentes. O Capítulo 7 descreve como reduzir as chances disso.

A reconciliação produz ruído. O drift check e a verificação de atrasos sinalizarão casos legítimos, como uma revogação que roda um minuto atrasada. Ajuste o período de carência para que um sinal signifique algo; uma verificação silenciada é pior que nenhuma, porque a equipe acredita estar coberta.

O break-glass tem um custo social. Quem declara um está dizendo aos contatos de segurança que o caminho normal falhou. Se a revisão parecer um julgamento, as pessoas hesitarão em emergências reais; se for uma revisão de design, não hesitarão. A revisão sem culpa é o que mantém o papel utilizável.

Quando não usar isto. Uma equipe de quatro engenheiros com um banco de dados de baixo risco e sem dados Restricted não precisa de pools de aprovação e de um job de reconciliação no primeiro dia. Um processo manual com um motivo, um limite de tempo, uma pessoa que confirma e um registro pode bastar por um tempo, se os grants forem removidos quando o tempo acabar e alguém conferir. Construa o padrão na ordem das suas falhas: primeiro a expiração, depois o registro, depois o aprovador, depois o alerta.

Erros comuns

- **Conceder o papel de teto a um login “só para esta tarefa”.** Toda sessão JIT passa então a ter o teto inteiro, e o escopo da requisição é decorativo. Conceda as triplas ao login de sessão e mantenha `human_jit_write` como um teto ao qual nenhum login pertence; o drift check sinaliza um membro.
- **Aprovar as triplas e não a instrução.** Uma lista de colunas é curta e fácil de ler, e aprová-la é aprovar uma capacidade. Vincule a requisição de escrita a uma correção revisada, derive as triplas dela e tome `approved_hash` do registro da correção.
- **Revogar a partir de uma lista separada.** Uma lista de tabelas fixada no código acabará esquecendo uma tabela que a requisição concedeu. Derive a revogação de `access_request_privilege`, o registro que conduziu o grant.
- **Confiar apenas no job de revogação.** É fácil parar de vigiar um job que roda todas as noites há um ano. Dê à sessão o seu próprio horário de término e execute uma verificação independente que compare grants com requisições.
- **Escrever a entrada de auditoria depois do grant.** Se o log estiver fora do ar, existe um grant que ninguém consegue ver. Escreva o registro, depois a entrada de log, depois os grants, e deixe que uma entrada de log que falhou interrompa o grant.
- **Aprovação automática de requisições de on-call.** Leituras não precisam de requisição e uma escrita precisa de uma segunda pessoa. Encurte a espera com um bom pool de aprovadores.
- **Tornar o break-glass longo o bastante para ser confortável.** Uma janela que nunca expira dentro de um dia de trabalho vira um papel de trabalho. Fixe o máximo no gateway e deixe que uma segunda declaração alerte de novo.
- **Deixar o conjunto de logging indefinido.** Quem revisa não consegue julgar uma sessão cujo registro nunca foi especificado. Nomeie o conjunto antes do primeiro uso.
- **Pular a revisão porque nada deu errado.** As revisões são onde o caminho normal é consertado. Dê um prazo à revisão e dispare um alerta quando ele passar.

- **Construir uma segunda porta para o dia ruim do gateway.** Um login de emergência fora do gateway contorna a sua identidade, o seu logging e a sua expiração. Invista primeiro em disponibilidade; se um segundo caminho for inevitável, dê a ele um responsável, um alerta e uma data de término.

Pontos-chave

- O acesso permanente custa mais nos dias em que não é usado, então a pergunta útil é quem precisa de acesso agora, para qual tarefa e como o sistema saberá que a tarefa terminou.
- `human_jit_write` é um teto e nunca é concedido a um login; uma requisição de escrita nomeia sua correção revisada (`fix_id`), as triplas derivam da correção, e o aprovador aprova a instrução, cujo hash (`approved_hash`) o runner precisa igualar.
- A revogação deriva do mesmo registro `access_request_privilege` que conduziu o grant, é imposta pelo horário de término da própria sessão, é apoiada por um job de revogação e é verificada por uma reconciliação independente que não confia em nenhum dos dois.
- As escritas exigem four-eyes: um aprovador que não seja nem o solicitante nem quem escreveu a correção, respondendo na tela do gateway depois de uma verificação multifator nova. Não existe aprovação automática para on-call.
- O break-glass é uma sessão `human_breakglass` autodeclarada, com uma janela máxima fixa, um conjunto de logging definido, um alerta na declaração e uma revisão obrigatória por alguém que não seja quem a usou. Não é um caminho de rotina nem um caminho de correção de dados, e seu único risco aceito é, hoje, um `UPDATE` sem limite e, no estado-alvo, quando o gateway do Capítulo 9 impõe um cap, linhas erradas dentro do cap.
- Um design sem um caminho rápido sancionado faz crescer um não sancionado, então a rota de emergência é projetada, registrada e ensaiada, e sua dependência do gateway é um custo que você nomeia.
- Onde o motor não consegue conceder em colunas, o fallback das tabelas de secrets mantém as colunas Restricted fora de toda requisição, e o aprovador aprova tabelas inteiras de colunas não Restricted.

Um grant, mesmo vinculado, diz o que uma sessão pode tocar e por quanto tempo, não com que cuidado a instrução por trás dele é verificada. O Capítulo 9 trata disso: o que o gateway faz com uma instrução antes de ela rodar. O Capítulo 10 então trata do trabalho em si: como executar uma correção de dados de modo que as linhas certas mudem, as erradas não, e uma segunda pessoa possa ver por quê.

Capítulo 9 — Execução mediada: análise de consultas e operações perigosas

A equipe de suporte da Ledgerly tem um assistente na sua ferramenta de chat. Sam pede que ele anule um lote de faturas duplicadas de um retry de cobrança, e o assistente, rodando sob a própria identidade, redige uma correção: um `UPDATE` em `invoices` com uma cláusula `WHERE` no rótulo do lote do retry. Jordan, da equipe de dados, revisa a instrução e a aprova. A instrução é SQL válido, o grant a cobre, o aprovador a leu, e o rótulo estava correto quando Jordan o verificou.

Mesmo assim, a instrução atinge mais do que as doze linhas que a correção nomeava. Um segundo job reutilizou o rótulo do lote dois dias antes, e nada no texto da instrução diz isso. Ninguém foi descuidado: Sam descreveu a tarefa com exatidão, o assistente escreveu o que lhe pediram para escrever, e Jordan revisou o que lhe foi mostrado. O que faltava era uma etapa entre “aprovado” e “commitado”, um lugar onde algo olhe para o que a instrução vai fazer com os dados e não apenas para o que ela diz.

Os capítulos 3 a 8 decidiram quem pode ter um grant, por qual caminho e por quanto tempo. Este capítulo acrescenta a parte Execução do modelo de cinco partes: a etapa em que o gateway trata cada instrução, em vez de entregar uma sessão e confiar em quem a detém. A pergunta que ele responde é esta: o que deve ficar entre uma instrução aprovada e os dados, quando a aprovação sozinha não consegue dizer o que a instrução vai fazer?

O problema e por que ele é difícil

Um session handout dá a uma pessoa um login e deixa o seu cliente falar diretamente com o banco de dados. Tudo o que vem depois é problema do cliente: o que é digitado, quantas linhas toca, se faz commit na hora. O

grant limita quais tabelas e colunas são alcançáveis, e nada limita o que cada instrução faz dentro desse alcance. No estado inicial, um cliente desktop envia o que for digitado e faz commit imediatamente.

A execução mediada inverte o arranjo. A instrução vai para um componente que não é o cliente da pessoa, e esse componente decide se ela roda, com quais limites e quando faz commit. O gateway do Capítulo 7 já está no caminho dos dados, então é o lugar natural para isso. A pessoa continua escrevendo a instrução, ou uma automação a escreve; o que muda é que nenhuma instrução chega aos dados sem passar pelo loop da próxima seção.

Isso é um espectro, não uma chave liga-desliga. A forma mais barata é um session handout com algumas configurações que o gateway impõe, como um timeout de instrução e um row cap. A forma seguinte verifica cada instrução antes de encaminhá-la. A forma mais mediada aprova cada instrução individualmente e a executa a partir de um worker, de modo que a pessoa nunca detenha uma sessão de banco de dados para a escrita. Algumas ferramentas aprovam cada consulta em vez de uma janela de tempo. No momento em que escrevo, o QueryProxy, por exemplo, faz uma segunda pessoa aprovar a instrução e um worker executá-la, de modo que a pessoa desenvolvedora nunca detenha a credencial do banco de dados. Isso troca a velocidade de uma sessão interativa por uma aprovação vinculada à instrução exata, e outras ferramentas de aprovação de consultas funcionam de modo semelhante.

A mediação é difícil por três razões. A primeira é que o SQL é expressivo: o mesmo efeito pode ser escrito de muitas formas, e se uma instrução é “perigosa” muitas vezes depende de dados que o verificador não leu. A segunda é que toda verificação tem um custo em latência, em recusas falsas e em pessoas que aprendem a contorná-la. A terceira é que um verificador em que se confia demais vira aquilo que ninguém revisa. A posição deste capítulo é que a análise de instruções reduz erros e não é a fronteira de autorização. Os grants são a fronteira. O trabalho do gateway é dificultar que a instrução errada mais comum rode por acidente e fazer com que toda escrita carregue um limite imposto por algo que não seja quem escreve.

O loop do gateway para cada instrução

Toda instrução que uma sessão envia, humana ou não, passa pelo mesmo loop. A ordem importa mais do que qualquer etapa isolada, então aqui está ele como pseudocódigo antes que as peças sejam explicadas.

```
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
```

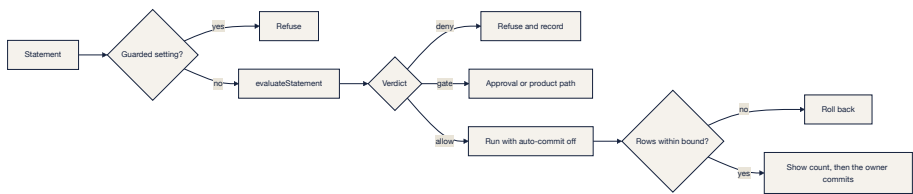
Guarded settings. O gateway abre cada sessão com seus limites aplicados: o timeout de instrução, um padrão somente leitura, o papel e o limite de conexões por tipo de sessão (Capítulo 5). Um cliente pode enviar um **SET** que levanta qualquer um deles, então o gateway mantém uma lista de

guarded settings e recusa uma instrução que altere uma delas. A palavra “não sobrescrevível” só é merecida onde um mecanismo a impõe. Um timeout definido como padrão de um papel pode ser levantado pela sessão, a menos que o gateway recuse o SET ou a plataforma o fixe, de modo que um limite que não tem nenhuma das duas coisas é um padrão e não uma garantia. O custo é que alguns clientes enviam essas configurações ao conectar e deixam de funcionar até que alguém as liste.

O row cap. Nas leituras, o gateway conta as linhas à medida que as transmite e fecha o cursor no cap, avisa a pessoa e registra o evento. O cap não diz que uma pessoa só pode ver tantas linhas ao longo de um dia, e uma pessoa determinada pode percorrer uma tabela em fatias. Ele diz que uma única consulta equivocada não puxa uma tabela inteira para um laptop. Extrações grandes e legítimas pertencem ao caminho de export do Capítulo 13 e à cópia de analytics mascarada do Capítulo 12, onde o tier decide quem aprova.

Auto-commit desligado para escritas. Uma sessão de escrita executa cada escrita dentro de uma transação explícita. Para uma pessoa ou uma migração, o gateway a abre; para uma correção de dados, o runner do Capítulo 10 a abre, e o gateway encaminha cada instrução da correção para essa única transação. De um modo ou de outro, o gateway conta as linhas afetadas em todas as instruções e as compara com o limite, enquanto a decisão de fazer COMMIT fica com o proprietário da transação: um gateway que fizesse commit de cada instrução de uma correção por conta própria quebraria a transação única do runner e faria do seu dry-run uma mudança real. Quando a contagem ultrapassa o limite, o gateway recusa a instrução e marca a transação como falha, de modo que o proprietário só possa fazer rollback. Essa é a forma barata de reversibilidade do Capítulo 1: um update que devia tocar uma linha e informa quatrocentas é interrompido enquanto desfazer ainda é de graça. Em uma sessão interativa, “mostrada” significa que uma pessoa vê a contagem e confirma. Para o runner e para a automação, “mostrada” significa que o gateway compara a contagem com o limite que a aprovação registrou e faz a transação falhar diante de qualquer excesso. O custo é a etapa extra e os bloqueios da transação, que continuam retidos enquanto a pessoa decide; uma pergunta de confirmação que ninguém responde segura um bloqueio de linha, então a pergunta tem um timeout que faz rollback.

Registro primeiro. O gateway escreve a intenção antes de a instrução rodar e o resultado depois. Não existe transação que abranja o log do gateway e o banco de dados, então a ordem é o que sustenta a garantia. Se a intenção não puder ser escrita, a instrução não roda. Se o resultado não puder ser escrito depois de um commit, o commit permanece, a entrada é enfileirada e a equipe de plataforma é alertada; um log que só funciona quando nada está errado não merece o nome. O Capítulo 14 trata de onde o registro vive.



Classificando uma instrução

O gateway decide o que fazer com uma instrução em `evaluateStatement`. Ele classifica a instrução pelo que ela faz e onde, consulta a matriz na seção depois da próxima e, para uma escrita que a matriz permite com um limite, encontra o limite e compara com ele a estimativa do planner. Toda recusa e todo gate vêm da matriz; a função não tem retornos antecipados próprios além dos do 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)

```

Três escolhas merecem defesa. Primeiro, uma instrução que o parser não consegue ler é recusada. Se o gateway executasse o que não consegue analisar, a análise valeria apenas para as instruções que nunca foram problema. Um cliente que envia lotes de instruções em uma única string quebra com esta regra, e isso é um custo a nomear na migração; a resposta é enviá-las uma a uma.

Segundo, a classificação olha para o efeito e não para o nome da pessoa ou da ferramenta. Um `DELETE` em uma migração e um `DELETE` em uma sessão JIT são a mesma classe, e a matriz decide o que cada lane pode fazer com ele. Ter um só classificador com uma só tabela, e não um conjunto de regras separado por caminho, é o que mantém a política pequena o bastante para ser revisada.

Terceiro, a estimativa do planner é comparada com a contagem esperada, mas apenas como uma recusa antecipada. O planner estima a partir de estatísticas, e as estatísticas ficam desatualizadas; uma estimativa pode ser

baixa demais tão facilmente quanto alta demais. A estimativa permite ao gateway recusar uma instrução claramente errada antes que ela tome bloqueios. O controle é a contagem real, comparada dentro da transação antes do COMMIT. A tolerância é um parâmetro que a equipe de plataforma define e registra, e começaria apertado e afrouxaria apenas quando as recusas falsas forem medidas, não supostas.

O que a análise estática não consegue ver

O texto de uma instrução diz pouco sobre o seu efeito nas linhas. Um parser consegue ver que falta um `WHERE`, que um predicado é sempre verdadeiro, que uma instrução toca uma tabela que não deveria ou que chama uma função. Não consegue ver quantas linhas um predicado casa hoje, que é o problema da abertura: um filtro válido, sobre um rótulo que agora casa mais linhas do que qualquer pessoa esperava.

Vários outros casos ficam fora do que a análise consegue resolver. Um filtro pode ser válido e ainda assim amplo, casando a maior parte de uma tabela por causa dos dados e não da sintaxe. Um join pode multiplicar linhas. Uma view ou uma função pode esconder uma escrita ou uma leitura mais ampla atrás de um nome de aparência inofensiva, e o gateway não consegue ver dentro do corpo de uma função que o papel pode chamar. Um efeito colateral disparado acontece depois da instrução e é invisível ao seu texto. E uma instrução pode estar exatamente certa na forma e errada no significado: as linhas certas pelo filtro, as erradas pela intenção, o que nenhuma ferramenta consegue julgar.

Ferramentas de código aberto como o QueryProxy verificam as instruções no envio, rejeitando um UPDATE ou DELETE cujo WHERE esteja ausente ou seja sempre verdadeiro e limitando SELECTs sem limite; no momento em que escrevo, sua própria documentação descreve esse parser como uma camada de defesa e não como a fronteira de autorização, que é o ponto desta seção, e outros gateways e ferramentas de aprovação adotam abordagens semelhantes.

O que decorre disso é uma divisão de trabalho. A análise pega os erros comuns de forma barata e cedo. Os grants (Capítulo 3) definem o que é alcançável, de modo que uma instrução que escapou ainda é limitada pelo papel. A verificação de contagem na transação pega o caso em que os da-

dos discordaram de quem escreveu. A aprovação, com uma segunda pessoa que lê a instrução e a contagem esperada, pega, algumas vezes, o caso em que a forma estava certa e a intenção estava errada. Nenhuma delas é completa, e o design se apoia na sobreposição entre elas e não reivindica nenhuma como a fronteira.

O mesmo limite molda o tier de export do Capítulo 13. O gateway calcula o tier de um export a partir das colunas de origem que a consulta lê, não dos nomes das colunas do resultado, e uma origem que ele não consegue resolver conta como o tier exportável mais alto; onde nem isso pode ser decidido, o export é recusado. É a mesma regra do parser, que falha fechado: o que a análise não consegue ler, ela não deixa passar.

A matriz deny, gate e allow

Cada classe de instrução recebe uma de três respostas em cada lane. **Deny** significa que o gateway recusa e registra a recusa. **Gate** significa que a instrução não roda como foi enviada: ela precisa de uma aprovação que ainda não tem, ou pertence a uma rota diferente. **Allow** significa que ela roda sob os limites do loop acima; onde uma célula diz “limite”, a escrita é permitida apenas dentro de um limite que vem da fonte que a célula nomeia, e a contagem real é comparada com ele antes do COMMIT. As colunas são os lanes que o livro descreveu, e as linhas são as classes de instrução que o classificador produz.

Classe de instrução	Sessão de leitura humana	Escrita JIT	Break-glass	Runner de correção de dados	Migração
Leitura, dentro do row cap	Allow	Allow	Allow	Allow	Allow
Escrita limitada, dentro da contagem aprovada	Deny	Allow, vinculada à correção (somente o runner)	Allow, vinculada a um row cap (nota obrigatória)	Allow, vinculada ao approved hash e à contagem	Allow, vinculada à regra de backfill
Escrita sem limite	Deny	Deny	Deny	Deny	Gate: aprovador nomeado

Classe de instrução	Sessão de leitura humana	Escrita JIT	Break-glass	Runner de correção de dados	Migração
Mudança de vínculo de proprietário	Deny	Gate: caminho de produto	Gate: caminho de produto	Gate: caminho de produto	Gate: aprovador nomeado
Escrita em coluna Restricted	Deny	Gate: caminho de produto	Gate: caminho de produto	Gate: caminho de produto	Gate: aprovador nomeado
Export	Gate: por tier	Deny	Deny	Deny	Gate: aprovador nomeado
Mudança de esquema (DDL)	Deny	Deny	Deny	Deny	Allow: lane revisado
Mudança de privilégios	Deny	Deny	Deny	Deny	Gate: revisão da plataforma
Escrita em ledger_entries ou audit_events	Deny	Deny	Deny	Deny	Gate: revisão da plataforma

Lê-la linha por linha mostra o design. A sessão de leitura humana não tem caminho de escrita, então toda classe de escrita é negada ali, e o único gate é o tier de export. Uma sessão de escrita JIT encaminha escritas apenas do runner, de modo que a sua célula de escrita limitada é um allow com vínculo e não uma porta aberta. O break-glass é o único lane em que uma pessoa escreve diretamente, e é por isso que a sua escrita limitada só é permitida sob um row cap e com uma nota, e por isso que o tamanho desse cap é a decisão de dimensionamento mais difícil do livro: uma emergência não é a tarefa planejada, então nenhum cap pode ser escolhido a partir da tarefa. O cap deve ser definido a partir do que o papel break-glass plausivelmente precisaria reparar e revisado após cada uso. Uma escrita sem limite é negada ali. O que resta é uma escrita que fica dentro do

cap e atinge as linhas erradas, um risco aceito que a revisão existe para expor, como o Capítulo 8 afirma; o Capítulo 8 também diz o que vale hoje, antes que um gateway imponha o cap.

A mudança de vínculo de proprietário merece sua própria linha. `UPDATE` de `customer_id` em `bank_accounts` ou `payment_tokens` passa pelo teto JIT, porque a coluna liga uma linha ao seu proprietário e não é `Restricted`. Mover a linha para outro cliente ainda muda de quem é a conta de repasse ou o token de pagamento. O classificador, portanto, a nomeia como uma classe própria, e todo lane exceto o de migração responde com um gate que a encaminha ao caminho de produto: o `admin-console` pede ao `payments-service`, uma segunda pessoa aprova, e a mudança é registrada em `audit_events`. Uma migração no lane normal a submete a gate para um aprovador nomeado, como o Capítulo 11 faz com qualquer instrução sinalizada, e o lane expresso a recusa; uma mudança de esquema tem pouco negócio reatribuindo propriedade, e, se uma reatribuição em massa for algum dia necessária, é uma funcionalidade de produto a construir e não uma instrução a executar. As escritas em colunas `Restricted` (`iban`, `account_holder_name` e `token_value`) seguem a mesma rota, e é por isso que nenhum lane permite que uma pessoa as escreva com SQL cru. O único caso de migração é mover colunas `Restricted` para as tabelas de secrets separadas do Capítulo 11, o que vai para um aprovador nomeado e não para uma recusa.

A coluna de migração corresponde ao que o Capítulo 11 descreve. Seu classificador sinaliza instruções para um aprovador nomeado ou para revisão da plataforma, e recusa apenas no lane expresso. Nesta tabela, as sinalizações aparecem como gates em que uma pessoa precisa dizer sim, e os `allows` são a mudança de esquema revisada e o `backfill` que fica dentro da sua regra. As tabelas somente de acréscimo seguem o mesmo princípio: uma migração que as toque vai para a equipe de plataforma, e os demais lanes a negam. Um segundo conjunto de regras para instruções proibidas seria uma segunda política a manter em sincronia com esta, então o pipeline de migração reutiliza o classificador e a matriz e acrescenta apenas as suas regras de lane.

A matriz é deliberadamente pequena. Nove classes e cinco lanes cabem em uma página, e uma política que cabe em uma página é lida. Cada nova classe ou lane acrescentada a ela é uma decisão que alguém precisa defen-

der, e a falha habitual é uma matriz que ganha uma exceção a cada incidente até que ninguém consiga dizer o que ela proíbe. Manteria a matriz em controle de versão, revisada como código, com a equipe de plataforma como responsável e a equipe de dados como revisora obrigatória das linhas de escrita.

Vinculando a aprovação à instrução

Uma aprovação que cobre “uma escrita em `invoices`” deixa passar qualquer escrita em `invoices`. A aprovação vinculada à instrução significa que a aprovação nomeia esta instrução, e o gateway recusa qualquer outra. O Capítulo 8 descreve a requisição e o Capítulo 10 o runner; esta seção é a regra que os conecta, enunciada uma vez.

Uma requisição de escrita carrega um `fix_id` e um `approved_hash`. O hash é o hash das instruções que foram revisadas, calculado sobre o conjunto inteiro quando uma correção tem mais de uma, e a sua origem importa mais do que a verificação. Ele vem do registro de merge da correção, ou do registro da requisição aprovada, e nunca da entrada do runner. Se o runner fornecesse o seu próprio hash, o runner estaria conferindo o próprio trabalho, e uma mudança na instrução chegaria junto com um hash correspondente. Um hash que se origina no registro que uma segunda pessoa aprovou é um hash de que alguém que não é quem escreve se tornou fiador.

```
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
```


O gateway verifica uma instrução por vez, mas o hash cobre o conjunto: ele carrega as instruções do registro de merge, confirma que o hash delas é o aprovado e aceita apenas a próxima instrução nessa ordem. O limite é a contagem de linhas esperada no mesmo registro, e não um número que a requisição carregue, de modo que o hash protege também a contagem. O commit permanece com a transação do runner, então uma correção com várias instruções é aplicada por inteiro ou não é aplicada.

Duas regras completam o vínculo. Em uma sessão de escrita JIT, o gateway encaminha escritas apenas da conexão do runner. O cliente da própria pessoa ainda pode ler dentro das triplas concedidas, mas uma escrita enviada dele é recusada, porque, de outro modo, o hash seria uma verificação que um cliente poderia contornar. Uma proteção só é um mecanismo se a sessão não puder escrever de nenhum outro modo. A segunda regra é o caso inverso: uma requisição JIT que contém apenas triplas `SELECT` não tem correção nem hash, então não há nada a que uma escrita possa ser vinculada, e a escrita é negada. Uma requisição de leitura não pode ser transformada em uma de escrita enviando uma instrução diferente.

Onde o gateway apenas intermedeia uma credencial e o cliente se conecta diretamente, nada disso é exigível, e o livro diz isso com clareza: o hash então registra uma intenção e não impede nada. O vínculo é uma propriedade do caminho dos dados e só existe onde o gateway está nele.

O caso do runbook é o que tem um modelo. O registro de merge de um runbook armazena o hash do modelo de instrução, e a requisição carrega apenas a lista de chaves. O runner substitui as chaves no modelo armazenado. Para esse caso, o gateway verifica o hash contra o modelo e não contra o texto substituído: confirma que a instrução enviada é o modelo aprovado apenas com a lista de chaves preenchida, e usa o número de chaves como limite, junto com as triplas da requisição. O gateway nunca aceita uma instrução ou um hash de quem quer que o trabalho seja feito.

Chamadores que não são pessoas

Jobs, scripts de runbook, bots de ChatOps e agentes de IA enviam instruções, e nenhum deles é uma pessoa. A regra deste capítulo é que eles passem pelo mesmo loop, como principais próprios. Não há caminho mais

leve para a automação, porque a automação é onde um erro se repete à velocidade de máquina.

Um chamador automatizado tem uma workload identity da plataforma e não a sessão de uma pessoa, e um padrão somente leitura. Ele abre a sua sessão sob o próprio principal, de modo que suas leituras ficam sujeitas ao mesmo row cap, timeout e guarded settings que as de qualquer outro. Se quiser escrever, segue a mesma rota de uma pessoa: nomeia uma correção, um aprovador que seja uma pessoa diferente da que acionou a automação a aprova, e suas escritas passam pelo runner com uma contagem esperada. A regra das duas pessoas do Capítulo 8 vale aqui também: quem pediu à automação que redigisse a correção não pode ser quem a aprova, e não existe aprovação automática por um chamador que não seja uma pessoa. Um agente que propõe uma instrução está propondo-a, e a aprovação é um humano lendo-a.

O registro nomeia as duas partes: a automação que enviou a instrução e a pessoa que a acionou. Esta é a metade da responsabilização. Um registro que dissesse apenas “o assistente fez” encerraria o rastro em uma máquina, e um registro que dissesse apenas “Sam pediu” esconderia qual ferramenta transformou a requisição em SQL. Os dois nomes são armazenados, e a revisão de acessos pergunta sobre ambos.

Um agente de IA acrescenta uma preocupação que os outros chamadores não têm: suas instruções são geradas a partir de texto, parte do qual ele não escreveu. Um ticket, uma linha de log ou uma mensagem de cliente podem acabar no contexto que produz a instrução. A defesa não é ler o texto com mais cuidado. É que os privilégios do agente não mudam com o que ele lê. Seus grants são os mesmos, suas escritas ainda precisam da aprovação de uma pessoa para a instrução exata, e uma instrução que chegue por uma rota que o agente não deveria ter, como uma escrita fora do runner, é recusada pela mesma matriz. Seja o que for que o agente tenha sido persuadido a propor, ele não pode aprová-lo, e não pode executá-lo sem o limite.

Para um agente somente leitura, os limites úteis são os comuns: um row cap apertado, um timeout curto, uma view ou uma cópia mascarada no lugar das colunas Restricted e um orçamento por principal, de modo que um loop que dê errado esgote a sua própria cota e não a compartilhada. Para um agente que escreve, o limite útil é que suas escritas sejam poucas e pe-

quenas, porque cada uma precisa de uma pessoa. Se isso tornar o agente lento demais para ser útil em escritas, a conclusão certa é que a tarefa deve virar uma funcionalidade de produto que o agente chama, não que a aprovação deva ser afrouxada.

Trade-offs e custos

A mediação custa tempo. Toda instrução passa por mais um componente, e uma escrita com gate espera por uma pessoa. Para leituras interativas em uma réplica, a latência acrescentada é pequena e quase invisível. Para escritas, é o ponto: o atraso é onde a segunda olhada acontece. Não o removeria para casos urgentes; tornaria o gate do lane break-glass rápido e ruidoso e o revisaria depois.

Custa compatibilidade. Um gateway que analisa instruções e recusa o que não consegue analisar quebrará alguns clientes, alguns drivers e alguns hábitos. Recusar instruções `SET`, strings com várias instruções e sintaxe incomum produz uma lista de coisas que deixam de funcionar. A lista é mais curta se for coletada antes do rollout, a partir dos registros de instruções das sessões existentes, do que se for descoberta por tickets de suporte. Reserve um período em que o gateway registre os seus veredictos mas recuse apenas os casos mais claros, e passe à imposição classe por classe à medida que as recusas falsas diminuam.

Custa um componente que precisa estar disponível. A verificação de instruções fica no caminho de toda escrita, e o Capítulo 7 descreve por que esse componente deve permanecer pequeno. Onde a verificação está fora do ar, o padrão que limita o dano é que as escritas param e as leituras na réplica continuam. Uma verificação que falha aberto ensina à equipe que a verificação é opcional, e a primeira indisponibilidade encerra a política.

Custa uma política a manter. A matriz, a lista de guarded settings, a tolerância e os row caps são todos parâmetros de que alguém é responsável. Uma política sem responsável deriva para o que o último incidente precisou. A matriz é o único artefato a manter pequeno, revisado e versionado.

Há também um custo que não se paga com engenharia: a análise dá uma falsa sensação de segurança. Uma equipe que tem um parser que rejeita `UPDATE` sem `WHERE` pode passar a tratar uma instrução que passou como uma instrução que não pode causar dano. A matriz diz o que o gateway

recusa, e a abertura deste capítulo mostra o que uma instrução aprovada ainda pode fazer. A verificação de contagem e a segunda pessoa existem para esse caso, e não devem ser reduzidas porque o parser é bom.

Erros comuns

Tratar o analisador como a fronteira. O analisador é uma camada que pega a instrução errada mais comum. Os grants decidem o que é alcançável, e um design que deixa o analisador carregar esse peso falha na primeira vez que uma instrução for válida e errada.

Deixar o runner ou o chamador fornecer o hash. Um hash que chega com a instrução prova apenas que a instrução corresponde a si mesma. O hash vem do registro de merge ou da requisição aprovada, e a verificação roda no gateway, onde o chamador não pode editá-la.

Auto-commit ligado em sessões de escrita. Com o auto-commit ligado, a contagem de linhas é um relatório posterior. Desligá-lo e mostrar a contagem antes do `COMMIT` é o que torna barata uma contagem errada.

Falhar aberto quando o parser não consegue ler uma instrução. Executar o que não pôde ser analisado faz a análise cobrir apenas os casos inofensivos. Recuse-a e mantenha uma lista curta dos clientes que precisam mudar.

Um caminho mais leve para a automação. Dar a jobs e agentes um atalho em torno do loop, porque são confiáveis ou porque estão com pressa, remove os controles onde o volume é maior. A automação passa pelo mesmo loop sob o próprio principal.

Uma matriz que ganha uma exceção por incidente. Cada exceção é uma decisão sem responsável. Mantenha a tabela nas classes e lanes que existem e, quando um caso não se encaixar, mude o design ou construa a funcionalidade.

Ler um row cap como um limite de exposição. Um cap limita uma consulta, não o que uma pessoa pode ler em um dia. Se a preocupação é a exposição, ela pertence ao trabalho de mascaramento e de tiers dos capítulos 12 e 13.

Pontos-chave

- A execução é o modo como uma instrução chega aos dados: quem a envia, se algo a inspeciona antes, o que limita o seu efeito e como ela sofre commit. Os grants não respondem a essas perguntas.
- A execução mediada coloca um componente entre a instrução e os dados. A forma mais barata é um punhado de configurações impostas; a mais forte aprova cada instrução e a executa a partir de um worker.
- Toda instrução, de uma pessoa ou de uma automação, passa por um só loop: guarded settings, avaliação, registro primeiro, um row cap para leituras e auto-commit desligado para escritas, com a contagem de linhas mostrada antes do COMMIT.
- A análise reduz erros e não é a fronteira de autorização. Os grants são a fronteira, e a contagem real de linhas verificada dentro da transação pega o que o parser e o planner não conseguem.
- Uma pequena matriz de classe de instrução por lane dá cada resposta como deny, gate ou allow. A mudança de vínculo de proprietário e as escritas em colunas Restricted passam por gate e são encaminhadas ao caminho de produto.
- A aprovação é vinculada à instrução: o `approved_hash` vem do registro de merge ou da requisição aprovada, nunca do runner, e em uma sessão de escrita JIT o gateway encaminha escritas apenas do runner.
- Jobs, runbooks, ChatOps e agentes de IA são principais com identidade própria, um padrão somente leitura e o mesmo loop. Seus registros nomeiam tanto a automação quanto a pessoa que a acionou.

O loop e a matriz dizem o que o gateway faz com uma instrução. Não dizem como escrever uma correção que valha ser aprovada: como provar a contagem, manter um snapshot, planejar o rollback e fazer uma segunda pessoa revisá-la. O Capítulo 10 trata desse trabalho.

Capítulo 10 — Consultas ad hoc e correções de dados

Imagine uma tarde de quinta-feira na Ledgerly. Um bug no `invoice-worker` foi corrigido e implantado, mas, enquanto estava no ar, deixou um lote de faturas com o status `sending`, embora os e-mails já tivessem saído. O suporte está perguntando sobre elas, os lembretes dos clientes estão pausados e Robin, da engenharia de backend e de on-call, enxerga o reparo em uma linha: voltar o status para `sent` nas faturas afetadas e em mais nada.

Robin abre uma sessão pelo access gateway. A sessão padrão é somente leitura, então a linha não pode ser executada. Robin precisa pedir acesso de escrita, e essa requisição tem de nomear tabelas e colunas. Outra pessoa precisa examinar a instrução antes que ela toque o primary. Enquanto isso, o relógio corre, e o reparo é tão pequeno que o processo ao redor parece maior que o trabalho.

Este capítulo acrescenta um passo ao estado-alvo: as escritas acontecem por meio de um procedimento revisado de correção de dados e de grants just-in-time dentro do teto `human_jit_write`. O Capítulo 7 deu às pessoas uma única porta, e o Capítulo 8 construiu os grants que a abrem por tempo limitado. Aqui perguntamos o que uma pessoa deve fazer depois que a porta está aberta: como ler, como escrever e como tornar o segundo tipo de trabalho seguro o bastante para ser feito com calma em uma tarde de quinta-feira.

No modelo de cinco partes do Capítulo 1, este capítulo trata da **Execução**. Robin já tem todas as credenciais de que o reparo precisa, então o risco não está em quem possui uma credencial. Está no que o grant autoriza, no que uma instrução fará ao ser executada, em quem a leu antes e em que evidência permanece. A pergunta deste capítulo é esta: qual é a maneira menor e mais revisável de uma pessoa alterar dados de produção, e quando a alteração deve deixar de ser tarefa de uma pessoa?

O problema e por que ele é difícil

Todo sistema que guarda dados reais chega, com o tempo, a um estado que o próprio código não consegue reparar. Um worker caiu entre dois passos. Uma migração deixou uma coluna preenchida pela metade. Um cliente foi mesclado a outro por engano. A aplicação não tem um endpoint para “consertar isto”, porque ninguém esperava que fosse necessário, e escrever o endpoint levaria mais tempo que o reparo. Então uma pessoa escreve uma instrução e a executa em produção. Isso é uma **correção de dados**: uma alteração em dados de produção feita por uma pessoa fora do caminho normal da aplicação, para reparar um estado que a aplicação não consegue reparar.

A primeira dificuldade é que a ferramenta perigosa e a ferramenta útil são a mesma. Uma sessão SQL que responde “quantas faturas estão travadas?” pode, com mais uma palavra-chave, alterar todas elas. No estado inicial do Capítulo 1, o cliente gráfico guarda o login de administrador compartilhado, faz auto-commit de cada instrução e executa o texto que estiver selecionado. A distância entre uma pergunta e uma alteração irreversível é uma palavra.

A segunda dificuldade é que a pessoa que faz o trabalho em geral está sob pressão e em geral tem razão. Um processo que trata cada correção como uma ameaça ensina as pessoas a contorná-lo, e o desvio é pior que o risco original: um grant permanente, uma senha compartilhada, uma cópia de produção em um laptop. Um processo que trata cada uma como rotina não ensina ninguém a olhar duas vezes, e a instrução errada, uma em cem, chega a todas as linhas. O caminho seguro tem de ser o fácil.

A terceira dificuldade é que a unidade de risco não é a pessoa, e sim a instrução, e uma instrução quase não carrega informação sobre o próprio tamanho. `UPDATE invoices SET status = 'sent' WHERE id = 'inv-0042'` e a mesma linha com a cláusula `WHERE` cortada diferem por alguns caracteres. Nada no texto diz “isto vai tocar nove mil linhas”; só os dados dizem isso, e os dados são lidos depois do fato, a menos que o processo faça a pergunta antes.

A quarta é que a resposta não pode ser só uma política. “Conte as linhas antes de atualizar” é lido uma vez e seguido até o primeiro dia movimentado, então precisa estar embutido no caminho: um runner que se recusa

a fazer o commit quando a contagem está errada e uma sessão que não consegue escrever de outra forma.

As consultas ad hoc têm um problema mais discreto. Uma **consulta ad hoc** é uma consulta que uma pessoa escreve e executa uma vez, fora de qualquer aplicação. Em uma sessão somente leitura ela não consegue alterar nada, mas pode ler muita coisa, pode ser cara para o banco de dados executar e seus resultados saem do sistema em uma captura de tela, uma planilha ou uma mensagem de chat.

O padrão: uma sessão somente leitura

Uma sessão humana começa como um login que é membro de `human_readonly`, conectado pelo access gateway a `ledgerly_replica_2`. Ela pode ler as tabelas que o papel cobre, com as colunas Restricted retidas e as colunas Confidential mascaradas onde o Capítulo 12 determina. Não pode escrever, e nada do que a pessoa digitar mudará isso.

A maior parte da investigação cabe dentro dessa sessão, e a pessoa não precisa pedir nada nem esperar ninguém. Defenderia esse padrão só pelo custo: um design em que a tarefa comum não exige aprovação é um design em que a aprovação para a tarefa incomum ainda significa alguma coisa.

Três hábitos mantêm seguros a réplica compartilhada e seus resultados. **Delimite a consulta:** o statement timeout no nível do papel (Capítulo 5) e o row cap do gateway (Capítulo 7) contêm o caso descontrolado, e selecionar contagens e amostras primeiro mantém pequenos os exports acidentais. **Selecione o mínimo que responde à pergunta:** `id, status, due_on` é melhor que `SELECT *`, em parte porque é mais rápido e em parte porque mantém as colunas Confidential fora do resultado, a menos que a pergunta precise delas. **Trate os resultados como cópias de dados:** um resultado colado em um ticket ou em uma planilha é uma cópia de dados de produção sob regras de acesso diferentes (o Capítulo 13 trata essas cópias como portas laterais), então cole identificadores e contagens, não linhas com conteúdo pessoal, e nomeie a consulta que os produziu.

Uma ressalva importa para a próxima seção. A réplica está atrás do primary por algum intervalo, em geral pequeno e às vezes não. Uma contagem feita ali é uma estimativa do que o primary guarda agora. Para inves-

tigar, isso basta. Para decidir quantas linhas uma escrita vai tocar, não basta, e o procedimento de correção de dados abaixo obtém a contagem definitiva no primary, dentro da transação que faz a escrita.

Escrever é um ato diferente

Uma leitura pode ser repetida, abandonada e ignorada. Uma escrita altera o que todos os outros principais enxergam, inclusive os serviços que estão em execução naquele momento e que agirão sobre o novo estado antes que alguém verifique. Um `status` atualizado por engano pode disparar um e-mail de lembrete; um `due_on` atualizado por engano pode mudar o que se diz a um cliente que ele deve. Por isso o livro dá à escrita um caminho próprio, com aprovação própria, limite de tempo próprio e registro próprio.

Esse caminho é o acesso just-in-time do Capítulo 8, com uma propriedade que importa aqui: uma requisição de escrita está vinculada a uma correção revisada. A requisição leva o `fix_id` da correção e um `approved_hash`, e suas triplas (privilegio, tabela, colunas) são derivadas da correção, não digitadas por quem pede. O máximo que uma requisição pode solicitar é o teto definido por `human_jit_write`. Três consequências moldam a maneira de escrever a correção.

Primeiro, **a requisição é construída a partir da correção**. Uma correção que atualiza `invoices.status` precisa de exatamente uma tripla: `UPDATE` em `invoices` para a coluna `status`. O guard da próxima seção lê antes de escrever, e essa leitura usa o `SELECT` simples em `invoices` que `human_readonly` já tem, então nenhuma tripla de `SELECT` é necessária. A exceção é uma correção que faz snapshot de uma coluna Confidential, como `billing_address` em `customers`: sob `human_readonly` a coluna é mascarada, e um snapshot de valores mascarados é inútil para um roll-back, então a requisição nomeia uma tripla de `SELECT` para ela e o aprovador a vê.

Segundo, **uma coluna Restricted nunca aparece na requisição**. `iban`, `account_holder_name` e `token_value` não podem ser nomeadas, e `INSERT` e `DELETE` em `bank_accounts` e `payment_tokens` não estão disponíveis, porque inserir ou excluir uma linha define ou destrói valores Restricted. Mais uma alteração fica dentro do teto e ainda assim não é uma

correção de dados comum: `UPDATE` de `customer_id` em `bank_accounts` ou `payment_tokens`. A coluna não é `Restricted`, mas mover uma linha muda de quem é aquela conta de pagamento ou aquele token de pagamento. A política de execução coloca essa alteração do vínculo de proprietário em `gate` e a encaminha ao caminho de produto descrito abaixo (Capítulo 9). Se uma correção precisa de qualquer uma dessas coisas, ela não é uma correção de dados.

Terceiro, **o aprovador vê a instrução, não apenas as triplas**. O Capítulo 9 expõe essa regra da aprovação vinculada à instrução e como o gateway a aplica. Para uma correção de dados, a tela de aprovação mostra a instrução, o filtro e a contagem esperada ao lado das triplas, e uma requisição de todas as colunas é uma requisição de outro tipo de confiança. Correções escritas com um escopo estreito são aprovadas mais rápido, que é o incentivo que você quer.

O registro da requisição para a correção das faturas travadas, na notação independente do motor do 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")
```

Em motores sem `grants` no nível de coluna, as colunas `Restricted` já ficam em tabelas separadas, fora do teto (Capítulo 3). Uma requisição JIT sobre a tabela principal é então um `grant` no nível de tabela, e o aprovador aprovou todas as colunas não `Restricted` dela, as `Confidential` inclusive. O `grant` deixa de limitar o alcance da instrução, então a própria instrução carrega mais da revisão.

Anatomia de uma correção de dados

Uma correção de dados é um procedimento curto, com ordem fixa. A ordem importa mais que qualquer passo isolado, porque cada passo existe para tornar o seguinte mais barato de acertar.

1. **Declare a intenção.** Uma frase: que estado está errado, que estado está certo e quais linhas estão no escopo. Se não cabe em uma frase, a correção provavelmente são duas correções.
2. **Conte as linhas afetadas com um `SELECT`.** “Quarenta faturas” e “nove mil” são conversas diferentes, e a contagem é o primeiro lugar em que um erro se mostra.
3. **Anote a contagem de linhas esperada.** A correção declara o número que espera alterar. Esse é o guard.
4. **Faça um snapshot de antes.** As chaves primárias e os valores antigos exatamente das colunas que a correção vai alterar, salvos em algum lugar que sobreviva à sessão.
5. **Execute a alteração dentro de uma transação.** Compare o número de linhas alteradas com o número esperado. Se diferirem, faça o rollback.
6. **Verifique uma pós-condição dentro da mesma transação.** Uma consulta que prova que o estado agora é o que a intenção dizia, por exemplo “nenhuma fatura da lista ainda tem o status `sending`”.
7. **Faça o commit e depois registre.** O commit é o ponto a partir do qual os outros principais veem a alteração. O estado de depois, lido dentro da transação, vai para o registro ao lado do snapshot de antes.

O pseudocódigo põe o guard no runner e não na memória da pessoa. Ele trata o banco de dados, o armazenamento de snapshots e o log de auditoria como sistemas separados, de modo que nenhuma transação cobre mais que o banco de dados. O runner recebe um id de requisição e nada mais; a primeira função decide se aquela requisição pode ser executada.

```

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

```

A segunda função a executa, e tudo o que ela faz é limitado pela primeira:

```

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)

```

Vários detalhes nesses blocos são deliberados.

O runner nunca fornece o próprio hash. Ele carrega a correção que a requisição nomeia e compara o hash das instruções dela com o `approved_hash` do registro da requisição, que o Capítulo 9 rastreia até o registro de merge. Digitar a instrução de novo muda o texto entre a revisão e a execução, e a revisão se prendeu ao texto anterior.

O snapshot é salvo antes da transação. Uma falha do armazenamento de snapshots então interrompe a correção sem nada alterado. O custo é que as linhas podem mudar entre o snapshot e a escrita: um worker pode

mover uma fatura de `sending` para `failed` nesse intervalo, e a chave ainda corresponderia ao filtro.

A transação lê as linhas de novo sob bloqueio. Ela compara o resultado inteiro, com as colunas que a correção altera, com o snapshot. Qualquer diferença no conjunto de linhas ou em um valor aborta a execução com `RowsChangedSinceSnapshot`, e quem escreveu a correção recomeça a partir de uma contagem nova. Em motores que não conseguem bloquear linhas por um filtro, a releitura ainda é executada dentro da transação; ela estreita a janela e não a fecha, e o que resta é a contagem esperada e a pós-condição.

O registro é escrito depois do commit. O armazenamento de snapshots e o log de auditoria são outros sistemas, então o estado de depois é salvo quando o commit já aconteceu. Essa é a ordem oposta à do grant JIT no Capítulo 8, em que um grant não pode existir sem sua entrada de log.

A ordem é defensável aqui porque o gateway da Ledgerly fica no caminho de dados (abaixo): ele encaminha as instruções do runner e registra a sessão, de modo que uma alteração com commit fica registrada mesmo que o append de auditoria falhe. A entrada passa por um retry e a equipe de plataforma é avisada. Em um gateway que apenas intermedeia credenciais não haveria log de sessão das instruções, e a entrada de auditoria teria de vir primeiro.

Um dry-run mostra o que uma contagem não mostra. A contagem é um `SELECT`, então não vê o que a instrução põe em movimento: uma trigger que escreve em outro lugar, uma cascata, uma regra. Um dry-run de verdade executa a alteração dentro de uma transação e faz o 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

```

O relatório contém o número de linhas, uma amostra dos valores de depois e a variação nas contagens de linhas das `watch_tables` do cabeçalho da correção. Ele exige uma requisição aprovada e os grants de escrita, então não pode existir enquanto o pull request está em revisão; é executado dentro da janela aprovada, mantém os mesmos bloqueios de linha enquanto roda e não enxerga efeitos fora do banco de dados, como um e-mail. Onde o custo de uma surpresa é alto, quem escreveu a correção o executa nos primeiros minutos da janela, anexa o relatório à requisição, e o aprovador o lê antes do passo de commit, ao preço de uma pausa.

Um guard só é um mecanismo se a sessão não consegue escrever de nenhuma outra forma. O gateway da Ledgerly fica no caminho de dados (Capítulo 7) e, em uma sessão de escrita JIT, encaminha escritas apenas da conexão do runner, de modo que a verificação de hash, a contagem e o snapshot não podem ser pulados. A transação continua sendo do runner: o gateway encaminha cada instrução para dentro dela e conta as linhas em toda a correção, mas nunca faz commit em nome do runner, e é por isso que uma pós-condição que falha desfaz todas as instruções. O Capítulo 9 explica como o gateway verifica cada instrução contra o registro de merge e onde a inspeção dele termina. O preço é que toda escrita JIT depende de o runner e o gateway estarem disponíveis. Com um gateway que só intermedeia a credencial, o runner é uma convenção, não uma proteção contra uma instrução sem cláusula `WHERE`.

Para onde vai o snapshot

O snapshot de antes é uma cópia de dados e precisa de um responsável e de uma expiração como qualquer outra. Um snapshot de valores de `status` em `invoices` é dado `Internal`. Um snapshot de uma coluna `Confidential`, como `billing_address` em `customers`, é `Confidential`, e o armazenamento precisa de regras de acesso pelo menos tão estritas quanto as da tabela de origem.

Guardaria os snapshots com o registro da correção, com um período de retenção definido pela equipe de plataforma: longo o bastante para que “esta correção estava certa?” possa ser respondida, curto o bastante para que o snapshot não se torne uma segunda cópia, sem governança, dos dados de clientes. Isso significa dias para status de faturas e talvez um ciclo de cobrança inteiro para um endereço de cobrança. Escolha um número, escreva-o e deixe-o expirar.

O plano de rollback

Um snapshot de antes não é um plano de rollback até que alguém tenha escrito como ele seria usado. O plano é uma ou duas frases no cabeçalho da correção, como “restaurar `status` a partir do snapshot para as mesmas chaves”, e deve ser possível transformar o snapshot em um script de correção de forma mecânica.

O rollback é, ele próprio, uma correção de dados. Passa pela mesma requisição e pela mesma revisão, porque outros principais podem ter agido sobre o novo estado nesse meio-tempo e restaurar os valores antigos pode sobrescrever atualizações legítimas. O runner de um rollback, portanto, verifica se os valores atuais ainda correspondem ao estado de depois registrado pela correção original e se recusa a tocar em linhas que mudaram desde então. Uma correção reversa que não sabe dizer “estas linhas seguiram adiante, trate-as separadamente” causa o próximo reparo.

Algumas alterações não têm rollback, e o cabeçalho deve dizer isso. Um e-mail, um webhook ou uma chamada a um provedor de pagamento não pode ser desfeito restaurando uma coluna. Para esses casos, a conversa acontece antes de a correção ser executada: que efeitos essa mudança de estado dispara, e os serviços que reagem devem ser pausados antes?

Duas tabelas não têm rollback no sentido comum, porque nenhum papel humano pode editá-las. `ledger_entries` e `audit_events` são append-only. Uma correção no ledger é uma nova entrada, emitida apenas pelo caminho de correção do `payments-service`, nunca por um login humano, de modo que uma correção que parece precisar de uma alteração ali é uma requisição para esse caminho, não para um grant. Uma trigger que rejeita todo `UPDATE` e `DELETE` nas duas tabelas faz a regra valer mesmo que um grant seja ampliado por engano. Isso tem um custo: `ledgerly_owner` ainda pode remover a trigger, então o ponto real de revisão continua nas migrações que tocam essas tabelas (Capítulo 11).

Um DELETE é o mesmo procedimento com um rollback mais difícil

A anatomia acima foi escrita para `UPDATE`. Suponha que um retry no `invoice-worker` tenha inserido linhas duplicadas em `invoice_lines` e que o reparo as remova. `DELETE` em `invoice_lines` está dentro do teto `human_jit_write`, e difere em três pontos.

O grant delimita a tabela e não as linhas, porque uma tripla de `DELETE` não tem colunas, então o limite vem do filtro, da contagem esperada e da revisão, e o filtro deve ser uma lista de chaves. O snapshot é a linha inteira, já que nada mais guardará os valores depois, e uma tabela com colunas Confidential torna o snapshot tão sensível quanto a tabela. O rollback é um novo insert, que é outra correção de dados com sua própria requisição. Uma tripla de `INSERT` nomeia suas colunas, então a requisição de reinserção decide se `id` e `created_at` estão entre elas. Se não estiverem, a linha restaurada recebe novos valores para ambos, e tudo o que referenciava os identificadores antigos não é restaurado. Se estiverem, o aprovador vê uma chave e um timestamp sendo escritos, o que nenhuma correção comum faz, e o que o motor faz com uma chave fornecida depende de como ele gera chaves. O cabeçalho informa qual escolha o rollback faz.

```

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")

```

Onde perder os identificadores não é aceitável, quem revisa pede uma proteção mais barata que o rollback: uma chave estrangeira que rejeita a exclusão enquanto outra linha ainda aponta para a linha, ou um reparo diferente, que não remova a linha.

Revisão: four-eyes, em um repositório

Uma correção de dados é código que roda uma vez, então recebe o controle em que a engenharia já confia: um pull request. A correção é um script em um repositório, com cabeçalho, revisor e histórico, e o histórico responde, seis meses depois, quem alterou um status e por quê.

```
// 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"
```

O campo `privileges` liga o repositório ao Capítulo 8. As triplas da requisição são copiadas dele, de modo que o aprovador olha a requisição e a correção lado a lado, e `loadApprovedFix` recusa uma requisição cujas triplas difiram das da correção. Uma instrução que toca uma coluna que seus privilégios não nomeiam falha em tempo de execução, que é o resultado pretendido; o `grant` é a segunda linha de defesa atrás da revisão.

O que o revisor lê

O trabalho do revisor é mais estreito que “isto é uma boa ideia?”.

- **A instrução corresponde à intenção?** Elas devem concordar nas tabelas, nas colunas e no filtro.
- **O filtro é específico?** Um filtro que seleciona por chave é o mais seguro; um filtro que seleciona por estado e um corte de tempo é aceitável se a contagem concordar; um filtro sem nenhuma restrição é rejeitado de imediato.
- **A contagem esperada é plausível e de onde veio?** Um revisor que vê `expected_rows: 40` pode pedir a saída da consulta de contagem no `pull request`. Um número que ninguém consegue rastrear é um número que ninguém verificou.
- **Os privilégios nomeiam apenas o que a instrução precisa?** Compare-os linha a linha.
- **A correção toca uma coluna `Restricted`, uma tabela `append-only` ou uma tabela fora do teto?** Se sim, não é uma correção de dados, e

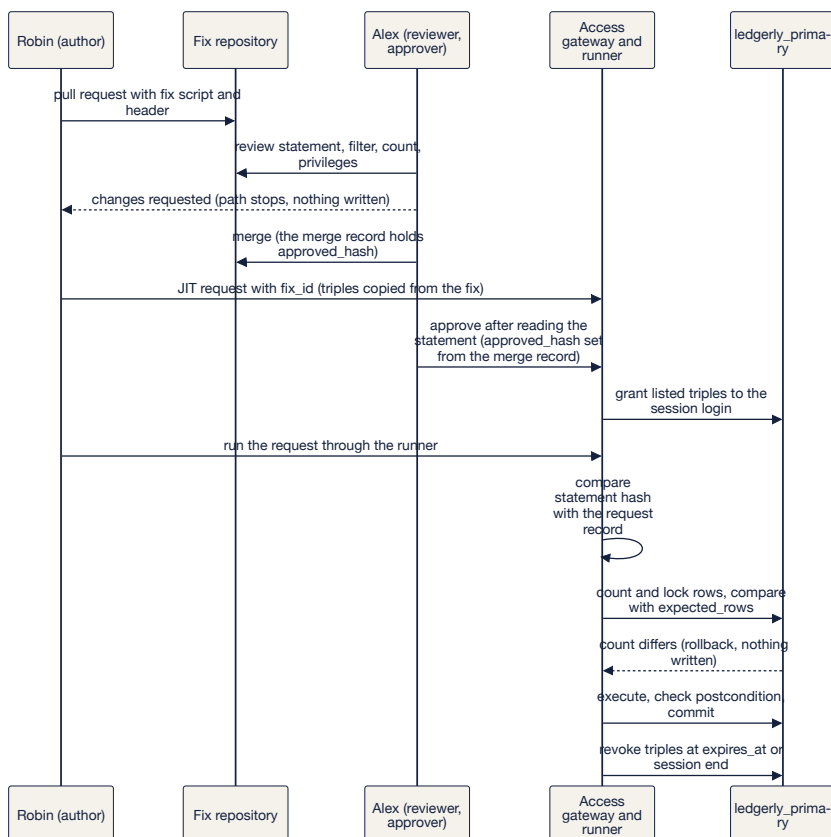
a revisão termina com uma indicação do caminho certo.

- **O que esta alteração dispara?** Lembretes, e-mails, jobs que leem a coluna. Esta pergunta pega os casos discretos.
- **A prévia concorda com a intenção?** O pull request contém a saída da consulta de contagem e uma prévia em `SELECT` das linhas e dos valores de depois; o relatório do dry-run é anexado à requisição mais tarde, e uma trigger ou uma cascata aparece ali e em nenhum outro lugar.
- **A frase de rollback é utilizável?** Para um `DELETE`, ela diz o que a reinserção não consegue restaurar.

A revisão é four-eyes no sentido do livro: uma segunda pessoa revisa antes que uma ação arriscada seja executada. O merge fixa o texto: uma alteração no script depois da revisão produz um hash diferente, e o runner a recusa. O aprovador da requisição JIT e o revisor da correção são dois atos. Em uma equipe grande preferiria duas pessoas, para que o aprovador confira o acesso contra a correção em vez de relê-la; em uma equipe pequena elas muitas vezes serão a mesma pessoa, o que é aceitável se nenhuma das duas for quem escreveu a correção. Quem escreve nunca aprova a própria correção nem o próprio grant.

A sequência

O diagrama mostra o caminho de uma requisição até uma alteração com commit. Procure os lugares em que o caminho pode parar sem nada escrito: a revisão, a comparação de hash e os guards do runner.



Quando a alteração é em uma coluna Restricted

O teto proíbe colunas Restricted em qualquer requisição JIT (o Capítulo 3 dá a razão), então toda uma classe de reparos não tem caminho SQL. Imagine que o `account_holder_name` de um cliente em `bank_accounts` foi digitado com um erro. Uma pessoa não consegue corrigir isso em uma sessão JIT, porque a coluna não pode ser nomeada em uma requisição. Uma sessão break-glass também não consegue: `human_breakglass` não tem privilégio de escrita em colunas Restricted, diga o que disser o processo.

O caminho é a aplicação. O `admin-console` pede ao `payments-service`, o único escritor de `bank_accounts`, que faça a alteração; o `admin-console` exige que uma segunda pessoa a aprove antes de o `payments-service` executá-la, e a alteração é registrada em `audit_events`. É um caminho de produto, não uma correção de dados: tem formulário, validação, campo de motivo, aprovador e registro, e roda com o papel do próprio serviço, não o de uma pessoa.

Isso custa mais que uma instrução SQL, e acho que deve custar. Esses valores são aqueles cuja exposição prejudica os clientes de maneiras difíceis de desfazer, e a aplicação é o único lugar em que vivem a validação e o conjunto completo de efeitos colaterais. O custo são minutos para quem precisa da alteração; a economia é que um reparo digitado à mão deixa de poder introduzir um número de conta bancária malformato. Um token errado em `payment_tokens` é igualmente substituído pelo `payments-service` pelo fluxo do próprio provedor, não remendado.

Break-glass não é uma rota de correção de dados

`human_breakglass` escreve onde o teto JIT não alcança, e é o único papel humano que lê colunas Restricted (seus privilégios exatos estão no Capítulo 3). No papel, parece uma forma conveniente de fazer um reparo que o caminho JIT recusa. Não é, e a razão é o que o papel não tem.

O break-glass é autodeclarado. Não tem aprovador, nem `fix_id`, nem runner, e suas salvaguardas são o conjunto de logs e a revisão obrigatória que o Capítulo 8 define. Uma ação de contenção que interrompe um dano em andamento, revisada depois, é para isso que o papel existe (o Capítulo 15 trata do lado do incidente). O reparo duradouro dos dados é outro ato e é sempre uma correção de dados: executado em uma sessão break-glass, ele pula o único controle que lê a instrução antes de ela rodar, em um papel em que um `UPDATE` sem guard de linhas é possível. O Capítulo 8 declara isso como um risco aceito, e encaminhar reparos pelo papel o tornaria rotina.

A segunda razão são os incentivos. Se o break-glass pode fazer um reparo, ele vira a via rápida, e o alerta dele deixa de significar qualquer coisa. Uma correção de dados que “precisa” de break-glass sinaliza outra coisa: o ciclo de aprovação noturno é lento demais, o teto não tem uma coluna de que a correção legitimamente precisa ou a correção pertence ao caminho da aplicação. Se um reparo não pode esperar, interrompa o dano com a menor ação disponível e execute a correção de dados pelo caminho normal quando a pressão passar.

Transformar uma correção repetida em um recurso do produto

Uma correção de dados é um sintoma: um estado é alcançável e a aplicação não consegue desfazê-lo. A primeira ocorrência é uma correção; a terceira é uma requisição de recurso que ninguém abriu. Manteria um diretório de scripts de correção, marcados pelo tipo de reparo, e, quando a mesma marca reaparecesse, alguém perguntaria se aquilo pertence ao `admin-console`. Não há nada preciso em “três vezes”: um reparo de dez linhas duas vezes por ano pode nunca justificar uma tela, enquanto um que roda toda semana, exige uma revisão a cada vez e poderia ser feito pelo suporte em um minuto é um recurso esperando para ser escrito.

Tome as faturas travadas da abertura e suponha que o estado se repita. A tabela de grants do Capítulo 3 já mostra o que `app_admin_console` pode fazer em `invoices`: `SELECT` e `UPDATE` apenas em `status`. A versão de produto é uma ação na página da fatura, disponível a um grupo restrito, que redefine o status de uma fatura, pede um motivo e grava a ação em `audit_events` com a identidade do principal humano. Ela carrega as próprias verificações: só faturas no estado travado, só uma por vez e um limite de quantas uma pessoa pode redefinir em uma hora.

A conversão desloca o risco da instrução, que paga por uma requisição, uma revisão e um runner a cada reparo, para o código, que é revisado, testado e versionado uma só vez. O suporte pode usar a ação sem uma sessão de banco de dados, então cada reparo que vira recurso é uma escrita recorrente que deixa de precisar de acesso 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()

```

O evento de auditoria e a mudança de status estão ambos em `ledgerly_primary`, então uma única transação cobre os dois; `audit_events` é o registro de negócio da aplicação, não o log de auditoria do Capítulo 14. No runner acima, o log de auditoria é um sistema separado e a ordem das operações é que carrega a segurança.

Nem todo reparo deve virar recurso. Um botão que qualquer pessoa pode apertar tem um perfil de risco diferente do de um script que duas pessoas revisaram, então reparos que alteram dinheiro, tokens ou dados bancários permanecem em um caminho four-eyes mesmo quando são frequentes. Um reparo só se torna uma ação de autosserviço se o pior caso dele for limitado pelo design da ação; caso contrário, ele se torna uma requisição com uma aprovação dentro do produto. O teste é se o reparo vai se repetir e se uma pessoa é hoje a única que consegue fazê-lo. Se as duas respostas forem sim, a sprint é um investimento e não um custo.

Runbooks para os reparos que você consegue prever

Algumas correções se repetem em tipo mesmo quando os detalhes mudam: reatribuir uma fatura ao cliente correto, reexecutar um export que falhou, reparar uma linha duplicada. Para essas, o artefato certo é um **run-**

book: um procedimento escrito e seguido para uma tarefa repetível. Ele tem o mesmo esqueleto de um script de correção, com os parâmetros em branco. Nomeia o sintoma que o dispara, as consultas de diagnóstico que o confirmam, o modelo da correção e os privilégios de que precisa, a origem da contagem esperada, os efeitos a verificar, o rollback e quem pode escrever e revisar. Diz também quando parar e escalonar: “se a contagem do diagnóstico excede o número deste parágrafo, este não é o problema conhecido”.

Um runbook também pode carregar uma pré-aprovação, para que um reparo previsível não espere por um aprovador à noite. Cinco condições decidem isso, e é o runner que as verifica, não a pessoa:

- o filtro seleciona por chave, uma lista de ids, e não por um estado e um corte;
- a contagem de linhas esperada é no máximo N, um número que o runbook define para o próprio reparo (digamos, 20 para faturas travadas; ilustrativo);
- as triplas são exatamente o conjunto fixo do runbook;
- o reparo não dispara nenhum efeito de saída, como registra o campo `effects` do runbook;
- a instrução é o modelo do runbook com apenas a lista de chaves substituída, e o hash do modelo foi aprovado quando o runbook foi revisado.

O registro de merge do runbook armazena esse hash do modelo. A lista de chaves é a única parte que a requisição carrega, e o runner a substitui no modelo armazenado e confere o resultado contra as triplas da requisição; ele nunca aceita um hash ou uma instrução de quem o chamou (o Capítulo 9 explica como o gateway verifica a instrução substituída). Se qualquer condição falhar, a requisição volta à revisão completa com um aprovador.

O custo é que uma execução pré-aprovada roda sem que ninguém leia a lista de chaves naquela noite, então o limite de linhas do runbook delimita uma lista errada, e alguém que não seja o runner lê o registro no próximo dia útil. Um runbook também envelhece: um runbook de uma tabela que mudou de formato desde então dá a uma instrução desatualizada a apa-

rência de revisada. Vincule cada runbook a uma data e a um responsável, e trate uma execução que falha como um sinal para revisá-lo, não como um incômodo a contornar.

Um UPDATE sem WHERE, analisado sem culpa

Considere a Ledgerly em seu estado inicial. Uma pessoa da engenharia se conecta com um cliente desktop que lembra a senha de administrador compartilhada e faz auto-commit. A tarefa são as faturas travadas. A instrução está escrita em duas linhas no editor: o `UPDATE` na primeira, o `WHERE` na segunda. O cliente executa a seleção destacada, e a seleção cobre só a primeira linha. A instrução roda, e todas as linhas de `invoices`, digamos nove mil, passam a ter o status `sent`.

Uma versão dessa história termina com alguém sendo mandado ter mais cuidado. Isso não ensina nada, porque a próxima pessoa na mesma cadeira enfrenta a mesma distância entre uma pergunta e uma alteração irreversível. A pergunta útil é o que, no design, deixou uma única instrução alcançar todas as linhas.

- **O login podia escrever em qualquer lugar.** O papel de administrador compartilhado é proprietário de todas as tabelas, então um erro não ficou confinado à tabela ou à coluna de que a tarefa precisava.
- **O cliente fez commit por conta própria.** Ninguém viu “nove mil linhas afetadas” antes de a instrução se tornar definitiva.
- **A contagem nunca foi pedida.** Nada exigia que a pessoa soubesse quantas linhas esperar, então o número impresso depois do fato não significava nada.
- **Ninguém mais tinha lido a instrução.** O único leitor era quem a escreveu, que leu a intenção de volta dentro do texto.
- **Não havia snapshot de antes.** Os status antigos se perderam, e o reparo do reparo não tinha ponto de partida.
- **O log dizia “o papel compartilhado”.** A pergunta “quem executou isto?” só tinha resposta pelo histórico do chat.

A recuperação começa por um backup, reconcilia o que aconteceu desde então e informa todos os que estão a jusante (o Capítulo 13 e o Capítulo 15 tratam dessa mecânica). O design fixou o custo antes de a instrução ser

digitada.

Agora percorra a mesma tarde no estado-alvo. Robin escreve a correção com seu cabeçalho e uma contagem esperada de 40, e Alex a revisa. A requisição JIT nomeia `UPDATE` em `invoices`, coluna `status`, e o grant dura uma hora. Se o filtro tivesse sido retirado do script revisado, o runner contaria nove mil contra os 40 esperados e se recusaria antes de qualquer instrução rodar. Uma instrução digitada no cliente da própria pessoa não seria encaminhada como escrita (a regra do gateway no Capítulo 9), e em uma sessão `human_readonly` simples o `UPDATE` não rodaria de forma alguma.

O guard não torna uma correção correta. Ele faz uma classe de erro ficar ruidosa: a instrução tocou um número de linhas diferente do que quem a escreveu declarou. Uma instrução que toca exatamente 40 linhas, mas as 40 erradas, passa, e é por isso que a revisão, a pós-condição e o snapshot existem como camadas separadas, e por isso que o Capítulo 9 diz o que a análise não enxerga. O guard também é tão bom quanto o seu número. Uma contagem esperada copiada da própria saída da execução ensina o runner a concordar consigo mesmo, então o número tem de vir de uma consulta que o revisor possa executar de forma independente, de preferência antes de a instrução ser escrita. Onde o motor tem grants no nível de coluna (Capítulo 3), um grant estreito limita até onde chegam os casos que o guard deixa passar.

Trade-offs e custos

O procedimento é mais lento que digitar uma instrução. Uma requisição, uma revisão, uma aprovação e um runner acrescentam minutos em um dia bom e muito mais quando o aprovador está dormindo. Em uma tarde de quinta-feira isso é aceitável. À noite, sob um alerta, é o custo que tenta as pessoas a recorrer ao break-glass, e merece atenção de design: um revisor na escala, a pré-aprovação do runbook acima para os reparos que se qualificam e um limite declarado para o tempo máximo que a equipe esperará por uma aprovação noturna, escrito antes da primeira noite em que for necessário.

A revisão pode virar reflexo. Uma aprovação dada em segundos acrescenta atrito sem reduzir a exposição; o Capítulo 8 faz o argumento completo. O sinal específico deste capítulo é o pull request: se os revisores nunca pedem a saída da consulta de contagem e nunca pedem uma mudança em uma instrução, ou não estão lendo nada ou só revisam correções perfeitas. As defesas são os itens do checklist que exigem um artefato: a saída da contagem e a prévia no pull request, o relatório do dry-run na requisição e uma comparação explícita da instrução com os privilégios.

Transações seguram bloqueios. O runner inicia a transação depois da revisão e a encerra em segundos. Um procedimento que a abre primeiro, “por segurança”, e deixa o humano decidir depois faz outros escritores esperarem, enche as conexões do pooler e deixa os timeouts do Capítulo 5 dispararem em trabalho não relacionado. Mantenha a decisão humana fora da transação.

Correções grandes precisam de lotes. Uma correção que toca milhões de linhas não pode rodar como uma única transação sem bloqueios longos e um log de transações grande. A resposta usual é dividi-la em lotes, digamos alguns milhares de linhas cada (um tamanho ilustrativo, não uma recomendação), cada um com sua contagem e seu guard, com um modo de lote do runner que repete as mesmas verificações. Uma falha no meio deixa os dados em um estado misto, então a correção tem de ser escrita para poder ser executada de novo: o filtro seleciona apenas as linhas ainda no estado antigo. Um reparo grande continua sendo uma correção de dados; só a primeira população de uma estrutura nova pertence a uma migração (Capítulo 11).

Snapshots e o repositório de correções precisam de um responsável. Um snapshot sem expiração trocou um risco por outro, e para colunas Confidential o segundo pode ser o maior. Um diretório de scripts de correção é um registro de como a produção se desviou, e sem um índice por marca ninguém procurará a ocorrência anterior de um problema.

Quando não usar tudo isto. Em uma equipe pequena, com um banco de dados que não guarda nada sensível, um pull request para cada mudança de status é mais cerimônia do que o risco justifica; contar primeiro e usar uma transação continuam valendo a pena. Em um banco de desenvolvimento ou de staging sem dados de produção, nada disso se aplica, e para

uma cópia reconstruída toda noite, como `ledgerly_analytics`, uma escrita errada custa uma reconstrução. O procedimento escala com o custo de errar, e o `primary` é onde esse custo é mais alto.

Erros comuns

- **Tratar a cláusula `WHERE` como o `guard`.** É tentador porque a cláusula é onde mora a intenção de quem escreveu. Mas uma cláusula é texto, e texto pode ser cortado, digitado errado ou executado em parte. O remédio é um `guard` que compara a contagem alterada com uma expectativa declarada e faz o `rollback` em caso de divergência.
- **Contar na réplica e atualizar no `primary`.** A réplica atrasa, então os dois números podem diferir por um bom motivo. O remédio é obter a contagem definitiva no `primary`, dentro da transação, e usar a contagem da réplica apenas para explorar.
- **Digitar de novo a instrução revisada.** O revisor aprovou um arquivo, e uma pessoa o copiou para um editor e mudou uma coisa “já que estava ali”. O remédio é um `runner` que compara o hash do script com o `approved_hash` do registro da requisição, junto com um gateway que encaminha as escritas da sessão apenas desse `runner`. Em um gateway que só intermedeia credenciais o `guard` é uma convenção, e a revisão tem de carregá-lo.
- **Pedir um `grant` amplo “por segurança”.** Isso faz da aprovação um julgamento de confiança em vez de uma verificação contra uma instrução, e amplia o que um erro pode alcançar. O remédio é pedir as triplas de que a instrução precisa e fazer uma segunda requisição se a situação mudar.
- **Usar `break-glass` porque o aprovador é lento.** O caminho é mais curto e a correção é urgente. O remédio é resolver a aprovação noturna lenta e reservar o `break-glass` para as emergências para as quais foi definido.
- **Escrever em uma coluna `Restricted` por uma sessão “especial”.** Atualizar um `iban` diretamente parece um favor ao cliente, e remove a validação, a aprovação e o registro da aplicação em um único passo. O remédio é o caminho da aplicação, mesmo que seja mais lento.
- **Salvar um `snapshot` que ninguém governa.** O `snapshot` protege contra uma correção ruim e se torna uma cópia sem governança de dados

Confidential em uma pasta compartilhada. O remédio é um responsável, uma regra de acesso igual à da tabela de origem e uma data de expiração.

- **Pular a frase de rollback porque a correção é pequena.** As correções pequenas são as que são revertidas com pressa. O remédio é um plano de uma linha, escrito antes da execução, que diz o que o snapshot restaura e onde ele não deve restaurar.
- **Deixar uma correção repetida como script para sempre.** O resultado é uma escrita recorrente que exige uma pessoa no primary a cada vez. O remédio é olhar periodicamente as marcas do diretório de correções e tomar uma decisão, mesmo que a decisão seja “ainda não”.
- **Revisar a correção da intenção em vez da correção da instrução.** Um revisor que debate se o reparo vale a pena pode nunca ler a instrução. O remédio é um checklist que faz as perguntas estreitas em ordem.

Pontos-chave

- Uma sessão somente leitura é o padrão certo para investigação, e só funciona se a tarefa comum não exigir aprovação.
- Escrever é um ato diferente de ler, então passa por uma requisição, uma revisão, um limite de tempo e um registro próprios. A requisição leva um `fix_id` e um `approved_hash` tirado do registro de merge, suas triplas são derivadas da correção e o aprovador vê a instrução.
- A ordem segura de uma correção de dados é contagem, snapshot, transação, guard de contagem de linhas, pós-condição, commit, e quem deve impô-la é o runner, não a memória da pessoa, com o gateway encaminhando as escritas da sessão apenas a partir do runner.
- Um guard faz uma classe de erro ficar ruidosa e não torna uma correção correta, e é por isso que a revisão, os grants estreitos e um snapshot de antes existem como camadas separadas.
- Uma alteração em uma coluna Restricted não é uma correção de dados: ela passa pelo `admin-console` até o `payments-service`, com um segundo aprovador e um registro em `audit_events`, e o break-glass não é uma forma de contornar isso.

- Uma correção manual repetida é um recurso de produto ausente, e cada uma que se torna uma ação do `admin-console` é uma escrita recorrente que deixa de precisar de uma pessoa no primary.
- Quando uma instrução alcança todas as linhas, a pergunta útil é o que no design permitiu isso, e a resposta é uma lista de guards ausentes, não uma pessoa.

As correções de dados mudam o que está nas tabelas; o próximo tipo de alteração muda as próprias tabelas. O Capítulo 11 se volta para as mudanças de esquema, para quem as executa e com qual identidade, e para o hábito de fazê-las com compatibilidade retroativa, de modo que o reparo de um dia não seja a indisponibilidade do seguinte.

Capítulo 11 — Mudanças de esquema e acesso para migrações

Imagine uma tarde de quinta-feira na Ledgerly. A equipe de faturamento quer uma nova coluna em `invoices` para que o imposto apareça separado do total, e a mudança é pequena: uma coluna, um valor padrão, um back-fill para as linhas antigas. Robin, que está de on-call nesta semana, revisa o pull request e o aprova. O pipeline executa a migração como `ledgerly_migrator`, o deploy termina e, por cerca de noventa segundos, toda requisição ao `billing-api` que toca uma fatura trava e depois falha.

Nada na migração estava errado, e ninguém fez nada fora do comum. Uma transação longa de lote do `invoice-worker`, deixada aberta enquanto esperava um serviço externo de e-mail, segurou um bloqueio na tabela, a migração entrou na fila atrás dela e toda instrução que chegou depois da migração entrou na fila atrás da migração. Alex, da equipe de plataforma, encontra a causa rapidamente. Vem então a pergunta mais difícil: a migração rodou com o poder do proprietário de todas as tabelas, rodou sem ninguém acompanhar, e a única coisa entre esse poder e o ledger era a revisão de um diff. Como as mudanças de esquema devem chegar ao primary, com qual identidade, e como evitamos que elas se tornem um buraco no modelo de acesso ou uma indisponibilidade?

Este capítulo acrescenta um passo ao estado-alvo: as migrações rodam como `ledgerly_migrator` a partir do pipeline, e nada mais tem o poder de alterar o esquema.

No modelo de cinco partes do Capítulo 1, uma migração é **Execução** com o limite mais largo que existe. O pipeline detém os privilégios do proprietário por design, então o risco não está em quem possui uma credencial. Está no que a migração está autorizada a fazer, em como suas instruções são executadas e em que registro sobrevive. Para migrações, isso significa uma lista curta de classes de instruções permitidas em cada lane (autoriza-

ção), um caminho que decide quando e como uma instrução roda, com um lock wait curto e um retry (execução), e um registro que o proprietário não consegue reescrever (responsabilização).

O problema e por que ele é difícil

Uma mudança de esquema é a coisa mais privilegiada que a maioria das equipes faz a um banco de dados em um dia rotineiro. A propriedade permite ao proprietário remover uma tabela, mudar o tipo de uma coluna, reescrever grants e, em muitos motores, contornar as restrições que valem para todos os outros. O Capítulo 3 deu esse poder a `ledgerly_owner`, um papel sem login, e deixou `ledgerly_migrator` assumi-lo apenas enquanto uma migração roda. Esse arranjo responde a quem é o proprietário do esquema. Não responde a quem pode iniciar uma migração, ao que uma migração em execução pode fazer nem ao que acontece com todos os outros enquanto ela roda.

Três riscos distintos estão dentro de uma única atividade.

O primeiro é a **autoridade**. Quem pode disparar o passo de migração do pipeline pode, na prática, executar instruções de nível de proprietário em produção. O que é permanente é a autoridade, no sentido que o Capítulo 3 descreveu: uma autoridade permanente de proprietário, exercida por meio de uma credencial de curta duração. Os limites sobre ela são procedimentais: a credencial vive apenas no pipeline, expira com o job e o pipeline é a única forma sancionada de usá-la. Esses limites são tão bons quanto os controles de acesso do próprio pipeline, o que faz do pipeline parte da fronteira de segurança do banco de dados, tenha alguém desenhado assim ou não.

O segundo é a **disponibilidade**. Muitas mudanças de esquema precisam de bloqueios, e o bloqueio é onde o dano acontece. Uma instrução que precisa esperar por um bloqueio não apenas espera: em muitos motores, as instruções que chegam atrás dela também entram na fila, de modo que uma migração que não consegue o bloqueio no primeiro segundo pode fechar a tabela para todos pelo tempo que a transação mais antiga à frente dela permanecer aberta. O tamanho da mudança quase não importa. Uma mudança de uma coluna pode ser tão disruptiva quanto uma reescrita se a transação errada estiver no caminho.

O terceiro é o **drift**. Um esquema que é definido por migrações e também editado à mão é um esquema que ninguém consegue descrever por completo. Um índice de hotfix adicionado a partir de um console em uma noite ruim é útil, e um ano depois é uma diferença entre ambientes de que ninguém se lembra, junto com grants que não correspondem a nenhum arquivo.

Cada um pode ser reduzido por design. Um papel de migração delimitado pelo pipeline, um lock wait que falha rápido, um formato de mudança que nunca precisa de um bloqueio longo e uma verificação que compara o esquema em produção com o esperado são coisas que você pode construir. Uma política que diz “tenha cuidado com as migrações” não está nessa lista.

Quem executa DDL, e com qual papel

DDL (data definition language) é a família de instruções que criam, alteram e removem objetos do esquema, em oposição às instruções que leem e escrevem linhas. A regra deste capítulo é curta: o DDL roda como `ledgerly_migrator`, a partir do pipeline, e nenhum papel de aplicação pode executá-lo.

Os papéis de aplicação não têm DDL

Nenhum dos papéis `app_*` do Capítulo 3 é proprietário de uma tabela nem pode criar, alterar ou remover uma. É fácil de dizer e um pouco caro de manter, porque muitos frameworks conseguem criar ou alterar tabelas na inicialização, e algumas equipes ativam isso porque elimina um passo do deploy. O custo dessa conveniência é que o serviço passa a precisar de propriedade, e um serviço comprometido ou confuso pode remover e re-conceder. Desligaria o gerenciamento automático de esquema em produção e o manteria nos bancos de desenvolvimento, onde ninguém precisa explicar uma tabela removida.

O efeito colateral visível é que um deploy que inclui uma mudança de esquema agora tem dois passos: a migração e depois o rollout do serviço. A ordem entre eles é uma questão de design que o padrão `expand and contract`, abaixo, responde.

O migrator e o proprietário

`ledgerly_migrator` tem um login que o pipeline usa; é membro de `ledgerly_owner` e não tem privilégios de dados próprios. A associação dá ao migrator os privilégios do proprietário, mas não faz do migrator o proprietário. Em muitos motores, um novo objeto pertence ao papel que estiver ativo quando ele é criado, de modo que uma migração que roda sob o próprio login do migrator deixaria suas tabelas pertencendo ao migrator. A regra é, portanto, a mesma em todos os motores: uma migração sempre muda para `ledgerly_owner` de forma explícita, porque a propriedade segue o papel ativo e não os privilégios herdados. Todo objeto que ela cria passa então a pertencer a `ledgerly_owner`, e a migração volta ao papel próprio do migrator quando termina. Em uma notação que não pertence a nenhum 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 })
```

Quatro detalhes importam. A credencial é emitida para este job pela workload identity do pipeline em tempo de execução (o padrão do Capítulo 4) e expira com o job, de modo que uma variável de pipeline copiada não pode manter por meses uma credencial de proprietário funcional e nada no repositório guarda uma. A mudança para o papel de proprietário é limitada às instruções de uma única migração. O registro de que uma migração foi aplicada é escrito na mesma execução, para que o banco de dados conheça a própria versão, da qual a verificação de drift abaixo depende. E a entrada externa é escrita antes da primeira instrução de proprietário, porque `recordApplied` vive em um banco de dados que o proprietário pode reescrever. O custo é que um armazenamento de auditoria indisponível interrompe a migração antes que ela comece. Essa é a ordem oposta à da correção de dados no Capítulo 10, em que o log de sessão do gateway já registra as instruções: aqui nada além do pipeline vê o que a sessão do proprietário faz, então o registro vem primeiro. Toda entrada de início recebe um fim correspondente, `migration-failed` ou `migration-applied`, e `isApplied` torna seguro um retry mesmo quando o último append de auditoria é o passo que falhou.

Em alguns motores, registrar a versão e executar o DDL podem compartilhar uma transação, e em outros uma instrução DDL encerra a transação ao redor dela. Saiba qual é o seu caso. Se uma falha pode deixar o esquema alterado e a versão sem registro, a migração precisa ser escrita de modo que executá-la uma segunda vez seja inofensivo.

Pessoas não usam o migrator

Pessoas nunca se conectam como `ledgerly_migrator`. Se alguém precisa de uma mudança de esquema, escreve uma migração, e o pipeline a executa. Isso soa como atrito, e é, mas compra uma propriedade útil: todo uso do poder do proprietário corresponde a uma execução do pipeline, que tem um commit, uma revisão e um log. Quando a credencial do migrator também está em um gerenciador de senhas da equipe, essa propriedade desaparece, e com ela a razão de ter o migrator.

O que o pipeline pode alcançar

Dar ao pipeline a credencial do migrator move um grande poder para um sistema que muita gente pode influenciar. Vale ser explícito sobre isso, porque é onde esse design pode dar errado sem alarde.

O que o pipeline pode fazer

Um job que executa `runMigration` pode criar, alterar ou remover qualquer objeto do esquema, mudar qualquer grant sobre objetos que pertençam ao proprietário e alterar as duas tabelas `append-only`, `ledger_entries` e `audit_events`. Nenhum papel de aplicação e nenhum papel humano pode editar linhas dessas tabelas, `human_breakglass` inclusive; a única exceção estrutural é que o proprietário, e portanto uma migração em execução, pode alterá-las. O controle dessa exceção não é um privilégio. É a revisão.

Assim, a pergunta sobre o pipeline é: quem pode fazê-lo executar alguma coisa? Isso inclui quem faz merge no branch de deploy, quem pode editar a definição do pipeline, quem pode adicionar um passo a um job que detém a credencial e qualquer pessoa que consiga ler o ambiente do job enquanto ele roda.

Delimitar o alcance

Delimitaria esse alcance de cinco maneiras, em ordem aproximada de quanto cada uma remove.

1. **Só um job detém a credencial.** O job de migração do ambiente de produção é o único lugar em que o login do migrator pode ser emitido, e ele é emitido pela identidade do job, não por uma variável que todo job pode ler. Builds de branches não revisados, jobs de teste e pipelines de pull request nunca a recebem.
2. **A definição do pipeline é revisada como as migrações.** Uma mudança no job de migração, nas permissões dele ou nas regras de branch que o disparam é uma mudança no alcance do proprietário e precisa do mesmo revisor que verificaria uma migração.
3. **O gatilho é um branch protegido.** O merge nele exige a revisão de uma pessoa que não seja quem escreveu a mudança, que é o controle four-eyes da Ledgerly para mudanças de esquema. Para migrações que

tocam grants, propriedade ou as tabelas append-only, o revisor deve vir da equipe de plataforma.

4. **A credencial é de curta duração.** Um job que falha e deixa um log com um secret deixa um secret que já expirou.
5. **Toda execução é registrada.** O pipeline escreve o que aplicou, com qual login, a partir de qual commit, em um armazenamento que o pipeline não consegue reescrever, e escreve a entrada de início antes de a migração rodar. O Capítulo 14 trata de onde esse armazenamento fica.

A própria revisão pode ser parcialmente mecânica. Uma pequena verificação pode ler a migração antes de uma pessoa e sinalizar as instruções que merecem um olhar mais atento:

```
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
```

No lane normal, a verificação não decide nada. Ela faz a atenção do revisor cair nas linhas que mudam acesso, seguram um bloqueio ou destroem dados, e torna visível a omissão de um grant. Uma verificação que bloqueia um merge a cada sinalização, sem forma de reconhecer uma, será contornada em um mês, então uma sinalização exige um aprovador nomeado e não proíbe a mudança. O lane expresso é o único lugar em que a verificação recusa: uma lista `refusals` não vazia faz o job falhar antes de rodar, e nenhum aprovador pode sobrepor isso. A próxima seção explica por quê.

O lane normal não recusa nada de forma categórica, e isso é uma escolha: uma lista rígida de instruções proibidas seria uma segunda política ao lado da matriz deny, gate e allow do Capítulo 9. Remover `ledger_entries` é sinalizado duas vezes, como mudança append-only e como mudança destrutiva, e a equipe de plataforma tem de dizer sim a ela.

Por que não uma pessoa com um ticket

Algumas equipes preferem que um DBA aplique cada migração à mão depois da aprovação. Para um banco de dados pequeno, com uma ou duas mudanças de esquema por mês, pode ser uma escolha razoável, e a questão do alcance do pipeline desaparece. O custo aparece quando a pessoa está ausente, quando a mudança precisa ser aplicada de forma idêntica em vários ambientes ou quando o passo manual se afasta do que foi revisado. A fraqueza do pipeline é a amplitude do alcance; a fraqueza do caminho manual é que nada garante que o que foi aplicado seja igual ao que foi aprovado. Inclino-me pelo pipeline com os limites acima, e reveria a escolha se a lista de acesso do próprio pipeline se mostrasse impossível de manter curta.

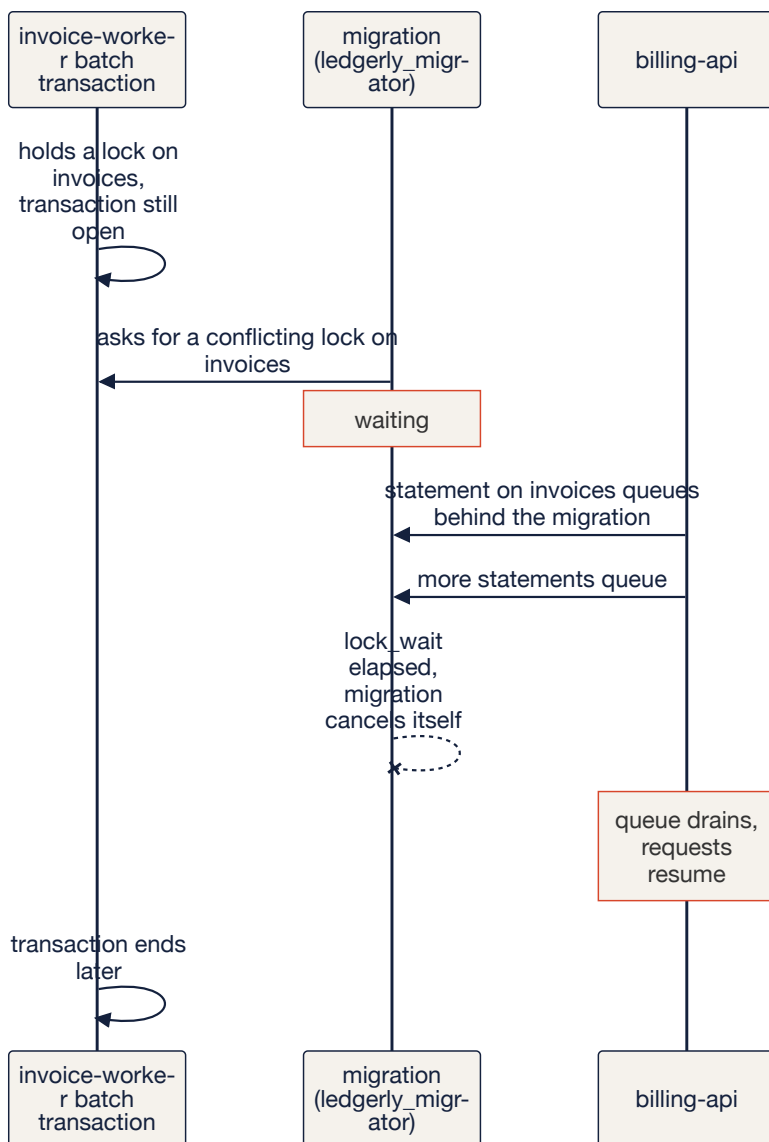
Bloqueios e o risco de disponibilidade

Voltemos aos noventa segundos da Ledgerly. O mecanismo por trás deles é comum a vários motores, e a defesa é construída a partir de um punhado de hábitos.

Como uma migração em espera fecha uma tabela

Uma migração que altera `invoices` precisa de um bloqueio que conflita com leituras e escritas comuns. Ela só consegue tomar esse bloqueio quando nenhuma transação que detém um bloqueio conflitante ainda está aberta. Se há uma transação longa aberta, a migração espera. Enquanto espera, novas instruções na tabela entram na fila atrás da requisição pendente da migração, porque deixá-las passar à frente a deixaria sem acesso. A tabela fica então efetivamente fechada até que a transação antiga termine ou a migração desista.

A sequência abaixo mostra a ordem dos eventos. Procure o ponto em que o `lock wait` dá à migração uma saída antes de a fila crescer.



O ponto é o autocancelamento. Sem um lock wait, a migração espera enquanto aquela transação de lote permanecer aberta, e o **billing-api** fica na fila por todo esse tempo. Com um lock wait curto, a migração falha depois de alguns segundos, a fila escoa e o pipeline pode tentar de novo mais tarde.

Um lock wait curto e um retry paciente

O Capítulo 5 definiu `lock_wait 3s` e nenhum statement timeout para `ledgerly_migrator`. Esse par é deliberado. Uma mudança de esquema pode precisar de muito tempo para fazer o trabalho, mas deve esperar muito pouco para começar. O retry pertence então ao 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
```

Algumas observações. Um lock wait que falha é um no-op limpo apenas quando nada foi confirmado ainda, e é por isso que uma migração carrega a marca `retrySafe`: ela é uma única unidade transacional ou o passo de bloqueio vem primeiro. Uma migração de vários passos, em um motor em que cada instrução DDL faz commit por conta própria, não sofre retry às cegas, porque a segunda tentativa encontraria o trabalho da primeira. Ela para, e uma pessoa decide. A pausa cresce com o número da tentativa para que tentativas repetidas não se tornem, elas mesmas, um padrão de carga. E o estado final, `MigrationBlocked`, é uma questão humana: descobrir qual transação está no caminho e decidir se espera, se a cancela ou se reagenda. Essa decisão envolve encerrar a sessão de alguém, o que é uma questão de acesso por si só e pertence ao Capítulo 15 quando acontece durante um incidente.

Instruções longas e mudanças online

Algumas operações mantêm um bloqueio enquanto rodam, como reescrever uma tabela grande para mudar o tipo de uma coluna. Para essas, o lock wait não é o problema; a duração é. Os motores diferem quanto a quais instruções são rápidas, e a resposta muda com a versão do motor, então verifique cada uma. Ferramentas de mudança de esquema online e construções de índice sem bloqueio evitam o bloqueio longo, mas criam e removem objetos, então precisam do mesmo alcance de proprietário que qualquer migração e rodam como o migrator, dentro do pipeline.

`classifyMigration` marca essas instruções como de bloqueio longo, e uma migração de bloqueio longo recebe uma janela agendada e uma pessoa nomeada acompanhando, escritas na descrição da migração. Isso não é uma falha da automação; algumas mudanças são eventos.

Expand and contract

A mudança de esquema mais segura é aquela com a qual o código antigo e o novo da aplicação conseguem conviver. O padrão para isso é o **expand and contract**: mudar o esquema em passos aditivos, mover a aplicação e só então remover o que não é mais usado. Ele importa para este livro porque cada passo é uma migração pequena com um bloqueio pequeno, e porque mantém os deploys reversíveis.

Considere uma mudança hipotética na Ledgerly, usada apenas para mostrar o formato: renomear `total_minor` em `invoices` para `amount_minor`. O esquema canônico mantém `total_minor`, e o exemplo supõe uma variante em que a coluna é renomeada. Os passos são assim:

1. **Expand.** Uma migração adiciona `amount_minor` como coluna anulável, com um bloqueio breve. O código antigo a ignora.
2. **Escrever em ambas.** As novas versões do `billing-api` e do `invoice-worker` escrevem a coluna nova além da antiga, de modo que versões antigas e novas podem rodar lado a lado.
3. **Backfill.** As linhas antigas recebem valores para a coluna nova, como descrito abaixo.
4. **Ler a coluna nova.** Se a versão se comportar mal, a coluna antiga continua correta, então o rollback é um deploy e não um reparo.

5. **Contract.** Depois de um período em que nada lê nem escreve `total_minor`, uma migração final a remove. Esse é o único passo destrutivo, e é controlado separadamente: revisado como destrutivo e nunca na versão que introduziu a coluna nova.

O expand and contract custa tempo e atenção. Uma mudança que poderia ser uma migração se torna quatro ou cinco, espalhadas por vários deploys, e durante os passos intermediários a aplicação carrega código para os dois formatos. Para uma equipe pequena, com um serviço e uma janela de manutenção, uma única migração pode ser a melhor escolha. Para a Ledgerly, com cinco serviços lendo as mesmas tabelas e nenhuma janela que o negócio aceite, o padrão é o que mantém o bloqueio curto e o rollback barato.

Backfills dentro da migração e correções de dados fora dela

O passo 3 é uma alteração em massa nos dados, e a mesma regra se aplicaria a dividir o total em líquido e imposto, com uma diferença: a tabela antiga não guarda valor de imposto, então esse backfill não poderia ser calculado e teria de deixar as colunas novas vazias até que uma correção de dados as preenchesse. O Capítulo 10 define uma correção de dados como uma alteração em dados de produção feita por uma pessoa fora do caminho normal da aplicação, para reparar um estado que a aplicação não consegue reparar, então o livro precisa de uma linha aqui: um backfill que pertence a uma mudança de esquema roda dentro da migração, como `ledgerly_migrator`; qualquer outra alteração em massa nos dados é uma correção de dados e segue o Capítulo 10.

O que faz um backfill pertencer à mudança de esquema?

- Seu único propósito é dar à estrutura nova seus primeiros valores, por uma regra declarada na migração.
- Ele é descrito na migração, revisado com ela e aplicado pelo pipeline.
- Todo valor que ele escreve poderia ser recalculado a partir de dados que já estão na tabela.

Se qualquer um desses pontos for falso, é uma correção de dados. A renomeação mostra onde fica a linha: todo valor escrito em `amount_minor` é o `total_minor` da mesma linha, uma regra que a migração declara e que um revisor pode conferir. Suponha que a equipe note, durante o backfill, que algumas faturas antigas têm um `total_minor` errado. Corrigir esses

totais não faz parte da mudança de esquema. Segue o Capítulo 10, com uma contagem das linhas afetadas, um guard sobre a contagem de linhas esperada e um segundo revisor, porque uma cópia que preserve um valor errado ainda é um backfill correto. O poder do migrator é grande, e ele não deve se tornar a rota para toda mudança incômoda.

Em uma tabela grande, o backfill roda em lotes, cada um uma transação curta sobre um conjunto limitado de linhas. Ele faz uma pausa quando o atraso da réplica cresce (Capítulo 6), desiste de forma visível se o atraso não se resolve e pode ser parado e reiniciado sem perder trabalho.

A mesma regra vale para as duas tabelas append-only. Uma migração pode adicionar uma coluna a `ledger_entries`, mas não reescreve as entradas existentes para preencher essa coluna com valores corrigidos; uma correção é uma nova entrada, emitida pelo caminho de correção do `payments-service`. Uma migração que toca essas tabelas é sinalizada pelo passo de classificação acima e vai para a equipe de plataforma revisar.

DDL de hotfix e drift

Toda equipe um dia enfrenta a noite em que um índice resolveria uma consulta lenta e o pipeline leva, digamos, vinte minutos. A pergunta é o que o modelo de acesso diz sobre o atalho.

Não há atalho pelo console, então construa um lane rápido

No estado-alvo, ninguém detém o login do proprietário.

`human_breakglass` escreve `INSERT` e `UPDATE` e não carrega DDL, e o superusuário é um instrumento selado que não se usa no trabalho diário (Capítulo 3). Assim, uma pessoa no primary não tem forma legítima de adicionar um índice à mão. Isso é deliberado, e desloca a pergunta para o pipeline: com que rapidez ele consegue executar uma mudança revisada quando a necessidade é urgente?

Daria ao pipeline um **lane expresso** em vez de um desvio. O lane usa o mesmo job, a mesma credencial e o mesmo registro; difere no tempo, não em quem precisa aprovar. O four-eyes permanece: um revisor que não seja quem escreveu a mudança ainda aprova, e quem escreveu não faz o merge sozinho. O que se relaxa é o período de espera agendado e, para uma lista curta de classes de instruções permitidas, a exigência da equipe de plata-

forma, com uma revisão de acompanhamento pela equipe de plataforma no dia útil seguinte. `classifyMigration` impõe a allowlist: criar um índice, remover um índice e adicionar uma coluna anulável são elegíveis. Uma instrução que muda grants, propriedade ou uma view, toca uma tabela append-only ou destrói dados não é, porque essas são as mudanças para as quais a revisão da plataforma existe, e uma migração que contém uma delas é recusada no lane e passa pelo lane normal. O Capítulo 15 descreve o acesso durante incidentes; o ponto aqui é que a urgência muda a velocidade do processo, não a identidade que executa o DDL nem a revisão que uma instrução perigosa recebe.

O custo é que o lane expresso é uma opção permanente de pular parte da cautela, e uma opção assim acaba sendo usada. Acompanhe com que frequência ele é usado e trate um número crescente como sinal de que o lane normal está lento demais, não de que a equipe ficou menos cuidadosa.

Drift

Drift é qualquer diferença entre o esquema que as migrações descrevem e o esquema que existe em produção. Ele entra por um hotfix aplicado à mão, por uma migração que foi editada depois de aplicada, por uma mudança de nível de proprietário feita por uma ferramenta de que ninguém se lembrava e por grants alterados fora da definição de acesso. O drift no esquema surpreende a próxima migração. O drift nos grants é uma mudança de acesso que nenhuma revisão viu.

Uma verificação agendada pode encontrar os dois 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")

```

Duas coisas tornam isso útil. O estado esperado é derivado das migrações e da definição de acesso do Capítulo 3, então há exatamente uma fonte de verdade para comparar. E a verificação cobre os grants, não só as tabelas, porque um grant que aparece fora de qualquer arquivo é o tipo mais importante de drift para este livro. Quando uma diferença é encontrada, a resolução é uma de duas: escrever uma migração que registre a mudança, se ela deve permanecer, ou escrever uma que a reverta, se não deve. Resolver o drift editando o registro do que foi aplicado esconde a mudança em vez de revisá-la.

Nota: Não “corrija” o drift aplicando uma instrução corretiva diretamente como o superusuário ou o proprietário. A instrução que remove a diferença é, ela própria, uma mudança não revisada em produção. Coloque-a em uma migração e deixe-a passar pelo lane.

Tabelas novas, colunas novas e os grants que as acompanham

O Capítulo 3 escolheu um padrão vazio para tabelas novas: uma tabela é legível por ninguém além do proprietário até que uma migração declare um grant. Esta seção é a outra metade dessa decisão, porque é nas migrações que a decisão se sustenta ou é esquecida.

Os grants vivem na migração

Cada migração que cria uma tabela declara seus grants no mesmo arquivo. Quem revisa vê então todo o efeito de acesso de uma mudança em um único diff, que é o argumento a favor do padrão vazio. Para uma tabela nova, `invoice_adjustments` (uma tabela ilustrativa, que não faz parte do esquema canônico), o arquivo fica assim. A coluna `reason`, de texto livre, é classificada como Confidential para o pior caso, como `invoice_lines.description`, de modo que os leitores que não são o proprietário da tabela recebem grants no nível de coluna que a deixam de fora:

```
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;
```

Esquecer um grant tem dois resultados possíveis, e a diferença é a razão do padrão vazio. Com um padrão vazio, o serviço que precisa da tabela falha na primeira consulta. Isso é ruidoso, é descoberto em um teste ou no primeiro deploy e se corrige com uma migração de uma linha. Com um privilégio padrão amplo, um grant esquecido é silencioso na outra direção: a próxima tabela é legível por quem o padrão nomear, e ninguém percebe porque nada falha. Uma falha ruidosa é mais barata que uma exposição silenciosa.

A falha só é ruidosa se algo executa a consulta. É por isso que incluiria no pipeline uma etapa de smoke test pós-migração que verifica se cada papel de aplicação consegue usar as tabelas que deveria usar. Os papéis de aplicação não têm login, então a etapa precisa de um mecanismo. Usaria um login de teste que é membro de cada papel de aplicação e muda para cada um em sequência, onde o motor oferece suporte; a alternativa é um login

de teste por papel. Qualquer das escolhas amplia o alcance do pipeline, porque um login que pode se tornar qualquer papel de serviço vale mais para um atacante do que qualquer credencial de serviço isolada. Por isso ele é emitido pela workload identity do job, expira com o job e é usado apenas nessa etapa. Em troca, a etapa pega o grant esquecido antes do rollout do serviço.

Colunas novas em tabelas que já têm grants

Uma coluna nova se comporta de modo diferente conforme o estilo de grant da tabela.

Estilo de grant da tabela	Uma coluna nova é...	Consequência
Grant no nível de tabela	legível por todo papel que detém o grant	Uma coluna sensível nova fica exposta de imediato
Grant no nível de coluna	legível por ninguém até ser nomeada	Uma coluna nova precisa receber um grant para ser usada

Para tabelas com colunas sensíveis, como `customers`, `bank_accounts` e `payment_tokens`, isso favorece grants no nível de coluna onde o motor oferece suporte, que é a escolha que o Capítulo 3 fez. A coluna nova começa então fechada, e a migração que a adiciona é onde a equipe decide quais papéis podem lê-la e se ela é Confidential ou Restricted (o Capítulo 12 trata da classificação). Uma migração que adiciona uma coluna a uma dessas tabelas sem nenhuma instrução de grant é um convite a uma pergunta, e a verificação de classificação pode fazê-la.

Onde o motor não tem grants no nível de coluna, vale a alternativa do Capítulo 3: as colunas Restricted vivem em tabelas separadas, como `bank_account_secrets` e `payment_token_values`. Uma nova coluna Restricted pertence a uma dessas tabelas, nunca à tabela principal, porque os grants da tabela principal são no nível de tabela e o teto do acesso JIT é traçado em torno dela. É fácil violar isso por acidente, já que adicionar uma coluna à tabela que todos já conhecem é o movimento natural, e é o tipo de erro que só um revisor que conhece a alternativa notará.

Views sobre tabelas sensíveis

Em muitos motores, uma view roda com os direitos do proprietário dela, o que significa que uma view sobre uma tabela com colunas Restricted pode revelar o que os papéis dos próprios leitores não poderiam. É por isso que `classifyMigration` sinaliza `CREATE VIEW` como uma mudança de acesso, por que a migração que cria uma passa pela equipe de plataforma e por que uma view sobre uma tabela sensível é julgada como uma mudança de acesso e não como uma conveniência.

Trade-offs e custos

O pipeline é um alvo de alto valor. Uma credencial que pode executar DDL de nível de proprietário, mantida por um sistema que muita gente pode editar, é uma concentração de poder. Um pipeline revisado não impede alguém com alcance de proprietário de fazer algo não revisado; ele torna esse algo visível depois do fato, o que é um controle mais fraco do que parece. Os limites deste capítulo reduzem o risco sem removê-lo. Se a lista de acesso do seu pipeline é longa e ninguém é responsável por ela, corrija isso antes de confiar em qualquer outra coisa.

As revisões trazem atraso e são tão boas quanto o revisor. Uma verificação four-eyes nas migrações atrasa as mudanças pequenas e pode virar formalidade, sobretudo quando o revisor vê, digamos, dez migrações por semana e todas são uma coluna. A verificação de classificação ajuda ao apontar as linhas perigosas. Ela não substitui a atenção. Uma revisão sempre aprovada em segundos é o sinal a observar, e o Capítulo 8 faz o argumento completo.

O expand and contract multiplica o número de mudanças. Uma mudança lógica vira vários deploys. Para uma equipe com uma janela de manutenção, um serviço e uma tabela pequena, pode não valer a pena. O padrão compensa quando muita coisa lê a tabela e ninguém pode pará-la pelo tempo de uma mudança.

Um lock wait curto faz as migrações falharem. Esse é o comportamento pretendido, e é um custo. Em uma tabela movimentada, uma migração pode precisar de várias tentativas, e alguém precisa decidir o que acontece depois da última. Se você definir a espera curta demais para a sua carga, o

pipeline enviará um page para a equipe de plataforma por ruído. Ajuste-a a partir de como são as suas próprias transações longas e diga isso no runbook.

As verificações de drift produzem achados que alguém precisa assumir. Uma verificação que alerta um canal que ninguém lê não encontra nada. Decida quem é responsável por cada classe de achado antes de ligar a verificação.

Quando não usar este design. Para um banco de dados descartável, um protótipo ou um projeto de uma só pessoa sem dados de clientes, um migrator com um papel de proprietário e um pipeline revisado é mais do que a situação pede. Execute as migrações a partir da máquina de quem desenvolve, mantenha o banco de dados de produção fora do caminho crítico e adicione o pipeline quando alguém além de você depender dos dados. O design deste capítulo é para o ponto em que o esquema é compartilhado por serviços e uma mudança errada custa algo aos clientes.

Erros comuns

- **Deixar o serviço criar ou alterar as próprias tabelas na inicialização.** Isso remove um passo do deploy e dá ao serviço a propriedade, que inclui o poder de remover e reconceder. Mantenha o gerenciamento automático de esquema desligado em produção e execute as migrações como um passo separado, com o migrator.
- **Guardar a credencial do migrator em um gerenciador de senhas da equipe “para emergências”.** É tentador porque o pipeline às vezes falha na pior hora. Isso transforma cada detentor em um proprietário e elimina a propriedade de que todo uso da propriedade corresponde a uma execução do pipeline. Construa o lane expresso em vez disso.
- **Executar uma migração sem lock wait.** A migração espera tanto quanto a transação mais antiga à sua frente, e toda instrução atrás dela espera também. Dê ao migrator um lock wait curto e coloque o retry no pipeline.
- **Esconder uma correção de dados dentro de um backfill.** Parece eficiente corrigir valores ruins já que a tabela está aberta de qualquer modo. A revisão da migração foi pensada para a definição de uma coluna, não para uma mudança nos valores dos clientes. Se a mudança

não é um primeiro preenchimento da estrutura nova por uma regra declarada, é uma correção de dados e segue o Capítulo 10.

- **Remover a coluna antiga na mesma versão da coluna nova.** Poupa uma migração e elimina o rollback. Mantenha o passo de contract separado, adiado e revisado como destrutivo.
- **Confiar em privilégios padrão para que tabelas novas funcionem sem uma linha de grant.** Um padrão amplo transforma um grant esquecido em uma exposição silenciosa. Mantenha o padrão vazio e deixe cada migração declarar seus grants.
- **Adicionar uma coluna Restricted a uma tabela principal em um motor sem grants no nível de coluna.** Os grants da tabela são no nível de tabela e o teto JIT é construído em torno deles. Coloque as colunas Restricted nas tabelas separadas de secrets.
- **Tratar um achado de drift como ruído.** Um grant que existe em produção e em nenhum lugar no código é uma decisão de acesso que ninguém revisou. Resolva-o com uma migração que o registre ou o reverta.
- **Editar uma migração depois de aplicada.** Os ambientes que executaram a versão antiga agora diferem dos que executam a nova. Adicione uma nova migração.

Pontos-chave

- O DDL roda como `ledgerly_migrator` a partir do pipeline, como o proprietário pela duração de uma migração, e nenhum papel de aplicação detém qualquer privilégio de DDL.
- O migrator detém uma autoridade permanente de proprietário, exercida por meio de uma credencial de curta duração: um job a detém, ela é emitida por workload identity, cada execução é registrada antes de começar e pessoas nunca a usam.
- Como o pipeline alcança o poder do proprietário, a lista de acesso e a definição dele são revisadas como uma migração; migrações que tocam grants ou as tabelas append-only vão para a equipe de plataforma, e o lane expresso aceita apenas uma allowlist de classes de instruções que `classifyMigration` impõe.

- Um lock wait curto com um retry no pipeline impede que uma migração bloqueada feche uma tabela, e ferramentas online e passos aditivos tratam as mudanças que de outro modo segurariam um bloqueio por muito tempo.
- O expand and contract mantém o código antigo e o novo funcionando juntos e faz do rollback um deploy, ao custo de mais migrações; um backfill que preenche a estrutura nova por uma regra declarada roda dentro da migração, e qualquer outra alteração em massa nos dados é uma correção de dados e segue o Capítulo 10.
- O drift no esquema ou nos grants é encontrado comparando o estado em produção com o que as migrações e a definição de acesso descrevem, e é resolvido por uma migração, não por uma edição silenciosa.
- Cada migração declara os grants do que cria e mantém os padrões vazios, de modo que um grant esquecido falha de forma ruidosa em vez de expor uma tabela em silêncio.

As mudanças de esquema são o único lugar em que um pipeline, e não uma pessoa, detém o maior poder sobre os dados. A próxima parte deixa de olhar para quem pode alcançar o banco de dados e passa ao que há dentro dele: o Capítulo 12 pergunta quais colunas merecem proteção, como classificá-las e como mascará-las ou retê-las sem quebrar o trabalho que precisa delas.

PARTE IV

Proteger os dados

Capítulo 12 — Dados sensíveis: classificação, mascaramento e minimização

Imagine uma segunda-feira na Ledgerly. Um cliente escreve para o suporte dizendo que um repasse não chegou e pergunta se a conta bancária cadastrada é a correta. Sam, do suporte, abre a página do cliente no `admin-console` e encontra o número completo da conta na tela. Ler o número de volta no chat é a forma mais rápida de responder, e um screenshot no ticket é a forma mais rápida de passar o caso a um colega. Ninguém na história faz nada de irrazoável, e no fim da tarde o número vive em um log de chat e em um anexo de ticket.

Duas mesas adiante, Jordan, da equipe de dados, está montando um relatório em `ledgerly_analytics` e percebe que a tabela `customers` ali traz nomes de contato, emails e telefones de que nenhum relatório precisa. Robin, que está de on-call, lê um log da aplicação para descobrir por que um lembrete falhou e encontra o endereço de email do destinatário dentro da mensagem de erro. Cada exposição é pequena, e as três têm a mesma forma: o dado chegou a um lugar onde ninguém havia decidido que ele deveria estar.

Este capítulo acrescenta um passo ao estado-alvo: as colunas sensíveis são classificadas, mascaradas ou retidas por padrão, e a cópia de analytics não contém dados Restricted em estado bruto. A pergunta que ele responde é o que cada pessoa e cada cópia realmente precisa ver, e como construir o sistema para que mostre isso e nada além.

No modelo de cinco partes deste livro (**Identidade, Grant, Caminho, Execução, Registro**), o capítulo trabalha em duas delas. O Grant decide quais colunas um papel pode ler, e a Execução decide em que forma um valor volta pelo caminho. Nada disso depende de como uma credencial chegou até uma pessoa; um login perfeitamente protegido que lê um número de conta completo já o expôs.

O problema e por que ele é difícil

Os controles dos capítulos anteriores decidem quem alcança quais tabelas e quais colunas. Eles não sabem por que uma coluna importa. Um grant em `customers` trata da mesma forma um flag de status e um email de contato, e o esquema chama os dois de `text`. Nada na definição de uma coluna diz que `invoice_lines.description` é texto livre que alguém pode ter preenchido com um nome, ou que `iban` é o tipo de valor cuja exposição não se desfaz rotacionando alguma coisa.

A segunda dificuldade é que as pessoas precisam de parte dos dados, e uma regra que diz que ninguém pode vê-los produz soluções alternativas. O suporte precisa responder a perguntas sobre contas de repasse. O on-call precisa descobrir por que um lembrete falhou. A equipe de dados precisa contar clientes por segmento. Quando o caminho autorizado não mostra nada, aparecem os não autorizados: screenshots, exports ad hoc, um grant permanente “até o fim do projeto”. A pergunta útil não é se uma pessoa pode ver um valor, mas qual forma do valor a tarefa exige: o valor inteiro, os últimos dígitos, um sim ou não, ou uma contagem.

A terceira dificuldade é que o dado não fica na tabela. Ele é copiado para um armazenamento de analytics, escrito em linhas de log, repetido em mensagens de erro, colado em tickets e enviado em exports. Todo controle no primary termina na borda do banco de dados, e o valor segue seu caminho. É por isso que a resposta precisa ser construída dentro do sistema e não pode ficar como política. Uma política que diz “trate dados pessoais com cuidado” não pode ser executada por um pipeline de migração, por uma carga agendada ou por um formatador de log. As decisões têm de ser escritas nas coisas que rodam: o esquema, os grants, a carga e o código de logging.

Os regimes de proteção de dados e de cartões de pagamento, como GDPR, KVKK ou PCI DSS, costumam fazer uma família de perguntas: quais dados pessoais você mantém, por quê, para onde eles vão, quem pode vê-los e por quanto tempo. Uma classificação mantida no esquema é a forma em que você consegue respondê-las. Interpretar qualquer regime cabe às pessoas de jurídico e de conformidade da sua empresa; este capítulo descreve apenas a engenharia que torna as respostas localizáveis. O catálogo é evi-

dência do que o sistema faz, não uma afirmação de que ele satisfaz algum regime; o Capítulo 14 volta ao tema de que evidência uma revisão de acessos consegue coletar.

Classificação que vive no esquema

A **classificação de dados** separa os dados em classes nomeadas, e cada classe determina o tratamento. A Ledgerly usa quatro.

Clas- se	O que signifi- ca	Exemplos da Ledgerly	Tratamento padrão
Res- tric- ted	A expo- sição prejudi- caria cli- entes ou a empre- sa de formas difíceis de desfazer	iban, account_holder_name, token_value	Retida de human_readonly; nun- ca em uma requisição just-in- time (JIT); alterada somente por um caminho da aplicação; ausen- te da cópia de analytics
Confi- denti- al	Dados pessoais ou co- merciais sobre cli- entes e contatos	contact_email, billing_address, provider_reference, invoice_lines.description	Mascarada ou retida para human_readonly; sem máscara somente em um SELECT JIT com aprovador; mascarada ou descar- tada na cópia de analytics, exce- to as dicas de exibição que exis- tem para ser mostradas (provider, expires_on; veja a tabela de decisões abaixo)
Inter- nal	Dados operaci- onais sem con- teúdo pessoal	amount_minor, status, due_on, account_code	Legível pelos papéis que preci- sam deles; copiada para analytics
Public	Seguro para publicar	Nenhuma entre as oito tabelas	Somente integridade

A linha Public está vazia de propósito. Ela existe para que “isto pode ser publicado?” tenha uma resposta diferente do silêncio.

Quatro regras tornam as classes utilizáveis. São a parte que escreveria antes de escolher qualquer ferramenta.

- **Classifique colunas, não tabelas.** `customers` contém um flag de status e um número de telefone, e os dois precisam de tratamentos diferentes. Apenas para fins de inventário, uma tabela assume a classe mais alta entre suas colunas.
- **Classifique o texto livre pelo pior caso.** Uma coluna em que as pessoas digitam acabará contendo tudo o que as pessoas digitam. `invoice_lines.description` é Confidential porque pode conter dados pessoais, e nenhum filtro consegue dizer quais linhas contêm.
- **Um valor derivado herda a classe de sua origem até que alguém argumente o contrário.** Um hash com chave de um `iban` é Restricted até que uma pessoa tenha argumentado que não é. `payment_tokens.last_four` é Confidential e não Restricted porque existe para ser exibido, e essa decisão foi tomada uma vez e registrada por escrito.
- **Desconhecido é Confidential.** Uma coluna que ninguém classificou é tratada como Confidential e marcada como provisória, de modo que o trabalho inacabado fique visível e o padrão seja o fechado.

Um catálogo no mesmo repositório das migrações

Uma classificação que vive em uma wiki se desvia, porque nada que altera o esquema lê a wiki. Manteria a classificação como um arquivo no repositório que contém as migrações, e a alteraria no mesmo pull request que adiciona a coluna. Alguns motores também permitem rótulos ou comentários em colunas; qualquer uma das formas funciona, desde que o pipeline consiga lê-la.

```
// 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
```

O catálogo torna possíveis duas verificações. A primeira roda contra cada migração, ao lado dos flags descritos no Capítulo 11. Diferentemente desses flags, que pedem a um revisor que olhe, esta bloqueia o merge, porque classificar uma coluna custa uma linha e a linha ausente é justamente o que todo o controle existe para pegar.

```
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)
```

Aqui, `inRestrictedHome` significa uma tabela em que os grants conseguem reter a coluna: uma tabela principal com grants em nível de coluna, ou as tabelas de secrets separadas do Capítulo 3 em motores que não oferecem esse recurso. Quando a pessoa que adiciona a coluna está em dúvida, a saída não é pular a verificação, mas classificar a coluna como Confidential com `reviewed_on` vazio e deixar que a próxima revisão de acessos resolva.

A segunda verificação roda de forma agendada e compara o esquema real com o catálogo, do mesmo modo que o Capítulo 11 compara o esquema real com as migrações. Uma coluna criada à mão em uma noite ruim aparece como um achado com responsável, e o achado se resolve por uma migração e uma linha no catálogo, não por uma edição silenciosa.

Minimização: os dados que você não mantém

Minimização é não coletar, não copiar nem guardar o que ninguém precisa. É o controle mais barato deste capítulo, porque um valor que nunca foi armazenado não tem máscara para manter, cópia para encontrar nem linha para submeter a redaction. Todos os outros controles aqui supõem que o dado existe; este pergunta se ele deveria existir.

Para cada coluna sensível, faria quatro perguntas, nesta ordem.

1. Alguma funcionalidade ou obrigação precisa deste valor? Se a resposta for “pode ser útil”, a resposta é não.
2. Poderíamos manter um substituto no lugar? A Ledgerly mantém `payment_tokens`, tokens emitidos pelo provedor que representam instrumentos de pagamento, e não os instrumentos em si. Essa única decisão remove do esquema toda uma classe de valores.
3. Poderíamos manter menos dele? Uma dica de exibição como `last_four` permite que o suporte e os clientes reconheçam um instrumento sem o instrumento.
4. Poderíamos mantê-lo por menos tempo? Os dados de contato de um cliente encerrado têm uma finalidade até que a última fatura seja quitada e os prazos de contestação tenham passado; depois disso, são mantidos apenas por um período curto e definido e então excluídos.

Duas decisões sobre onde os dados vivem importam mais do que qualquer regra de retenção. A primeira é que **tabelas append-only não devem conter dados pessoais**. `ledger_entries` contém `amount_minor` e `account_code`, ambos Internal, e é por isso que uma correção pode ser um novo lançamento: nada na tabela precisa jamais ser apagado para atender a uma requisição sobre uma pessoa. `audit_events` refere-se ao seu sujeito por um `subject_id` opaco pelo mesmo motivo. A segunda é o texto livre. Um campo estruturado com um conjunto fixo de valores é clas-

sificável e mascarável; uma caixa de anotações não é nem uma coisa nem outra, então, onde uma funcionalidade pode usar um código, um status ou uma referência de produto no lugar de uma frase, escolheria isso.

Retenção é um job, não uma frase em uma política. Algo precisa rodar de forma agendada, encontrar as linhas que passaram do período e excluí-las ou anonimizá-las, e esse algo precisa de privilégios estreitos próprios, de entradas de auditoria próprias e de um responsável. Não sei qual é o período certo para os seus dados, e este livro também não sabe. Ele depende do que seus contratos, seus reguladores e seu próprio processo de contes- tação exigem, e o jurídico precisa responder a isso antes que uma pessoa de engenharia escreva o job. O que a engenharia pode dizer é que bac- kups, exports e a cópia de analytics também guardam dados antigos, e uma exclusão que alcança apenas o primary não terminou. O Capítulo 13 volta a isso.

Mascaramento: dinâmico e estático

O **mascaramento** substitui ou obscurece um valor para que o valor real não fique visível enquanto o formato continua utilizável. A **redaction** é a versão bruta: o valor é removido de uma saída ou de um log. Os dois tipos de mascaramento diferem no ponto em que atuam.

	Mascaramento dinâmico	Mascaramento estático
O que muda	O que uma consulta retorna	A própria cópia
Quando	Na hora da leitura, por leitor	Na hora da carga, uma vez
Valor bruto ainda armazenado	Sim, no mesmo armazenamento	Não na cópia
Uso típico	<code>human_readonly</code> em uma réplica	A cópia de analytics, dados de staging
Como falha	Comportamento do motor, filtros, grants diretos	Uma cópia desatualizada ou aparada demais

Mascaramento dinâmico para pessoas

`human_readonly` é o padrão com que toda sessão começa, então o seu mascaramento é o que a maioria das pessoas encontra. A notação abaixo é ilustrativa, porque os motores expressam o mascaramento de formas mui-

to diferentes: alguns oferecem uma máscara na coluna, outros um privilégio de remoção da máscara, outros uma view fixa por tabela.

```
// 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;
```

A escolha da função decorre da finalidade da coluna. Uma técnica não é melhor nem pior em si; cada uma tem seu lugar e sua forma de falhar.

Técnica	O que o leitor recebe	Adequada para	Onde deixa de ajudar
Parcial	Os primeiros ou os últimos caracteres	Uma tela, um resultado colado	Valores curtos mostram a maior parte de si mesmos
Redaction	Um marcador fixo ou nada	Endereços, texto livre	Não consegue confirmar nada
Pseudônimo com chave	Um substituto estável	Joins entre cargas	Rotacionar a chave quebra os joins
Generalização	Um valor mais amplo, como um país ou um mês	Analytics	Colunas combinadas podem identificar uma pessoa (Capítulo 13)

Um hash simples não está na lista. O hash de um valor tirado de um conjunto pequeno o bastante para que todos os candidatos possam ser testados, como um número de telefone, pode ser revertido testando-os, de

modo que um pseudônimo precisa de uma chave mantida fora do banco de dados.

Nem toda coluna Confidential é mascarada. Uma coluna que existe para ser mostrada continua legível: `payment_tokens.provider`, `last_four` e `expires_on` são as dicas de exibição que permitem reconhecer um instrumento, e mascará-las apenas empurraria as pessoas para o valor Restricted que elas substituem. Os campos de contato, o endereço de cobrança e `provider_reference` identificam uma pessoa ou uma transação e são mascarados. `invoice_lines.description` é retida.

Isso dá a regra de que o Capítulo 8 depende. `human_readonly` já lê, então uma tripla de SELECT JIT só é necessária para uma coluna Confidential que o papel mascara ou retém, e o que ela acrescenta é essa coluna sem máscara. As colunas Restricted nunca aparecem em uma tripla JIT, então nada muda para elas. A requisição e a tela do aprovador devem, portanto, trazer uma linha explícita, “Colunas sem máscara: `contact_email`”, separada da lista de tabelas, para que o aprovador veja quais máscaras a requisição remove e não apenas quais tabelas ela toca.

Aviso: Verifique como o seu motor resolve um login que lê uma coluna por meio de `human_readonly` e também possui um grant direto na coluna. Se a máscara prevalecer, um SELECT JIT não mostra nada de novo e a aprovação não significa nada. Se os valores brutos prevalecerem para todo leitor, a máscara é decoração. Teste os dois casos em uma cópia antes de confiar em qualquer um.

Dois limites devem ficar ao lado da técnica. Primeiro, em muitos motores uma máscara se aplica ao que uma consulta retorna, não ao que suas condições comparam, de modo que um leitor autorizado a filtrar por uma coluna mascarada pode descobrir algo sobre seus valores por meio de contagens e correspondências. Trate o mascaramento dinâmico como proteção contra exposição casual, como uma tela, um screenshot ou um resultado colado. Segundo, mascarar uma saída não faz nada pelo valor armazenado, que permanece em todo backup, réplica e dump do primary.

O primeiro limite gera uma regra de decisão para a inferência. Pergunte com que baixo custo um leitor poderia recuperar o valor filtrando: um punhado de sondagens de igualdade em `contact_email` é barato, e uma

contagem de clientes por prefixo de telefone mascarado também é barata. Onde é barato e o leitor não deve conhecer o valor, retenha a coluna em vez de mascará-la, ou negue predicados sobre a coluna mascarada para esse papel. Onde é caro, como sondar um valor livre longo um caractere por vez por meio de milhares de instruções registradas, o mascaramento mais o registro das instruções é um controle razoável.

Mascaramento no stream de resultados

Alguns caminhos de acesso não podem contar com o motor para mascarar. Um gateway que fica entre a pessoa e o banco de dados pode aplicar regras ao próprio stream de resultados, antes que as linhas cheguem à tela ou a um arquivo de resultado. Isso ajuda quando o motor não tem recurso de mascaramento, e as regras podem ser as mesmas para todo motor por trás dele.

Uma regra de stream depende de saber de que coluna veio um valor, e um alias como `SELECT api_key AS k` derrota regras que casam pelo nome da coluna. As ferramentas que mascaram resultados, entre elas o QueryProxy, acrescentam regras que casam também pelo valor, o que é uma segunda camada, nunca um substituto do grant. No momento da escrita, ele próprio documenta essa limitação das regras baseadas em nome, e outras ferramentas adotam abordagens semelhantes. A consequência para a Ledgerly é que uma máscara de stream fica atrás dos grants de coluna do Capítulo 3, e uma coluna que um papel nunca deve ver é retida pelo grant, não deixada a cargo de um padrão.

Nota: Em motores sem grants em nível de coluna, a Ledgerly usa o fallback do Capítulo 3: as colunas Restricted ficam em `bank_account_secrets` e `payment_token_values`. Mascarar `human_readonly` significa então, em geral, uma view mascarada fixa por tabela, criada uma vez por uma migração e pertencente a `ledgerly_owner`, com o papel tendo `SELECT` na view e não na tabela base. Um grant JIT em nível de tabela na tabela base mostra em estado bruto todas as colunas não Restricted, as Confidential inclusive, e o aprovador aceitou exatamente isso. Trata-se de uma view fixa, não de uma view criada por requisição, e é um mecanismo de mascaramento para colunas Confidential. Não é um segundo fallback para as Restricted: o único fallback para essas são as tabelas de secrets.

Mascaramento estático para a cópia de analytics

A cópia de analytics é onde o mascaramento é mais barato e mais útil, porque o valor bruto pode ser deixado totalmente de fora da cópia. O relatório de Jordan precisa de segmentos de clientes, contagens e valores. Ele não precisa de nomes, emails nem telefones, então, para a maioria das colunas Confidential, o tratamento correto é `drop`, e uma transformação é reservada às poucas que os analistas comprovadamente usam.

A carga é um job agendado, não um serviço, e roda sob um login próprio e o papel `app_analytics_load`, como qualquer outro principal na revisão de acessos. O papel tem `SELECT` apenas nas colunas que o catálogo não marca como Restricted, de modo que o grant é a primeira proteção e o catálogo a segunda. Elas não são independentes, porque o grant é derivado das marcações Restricted do catálogo: uma coluna classificada por engano como Internal passa nas duas. O que o grant acrescenta é que uma coluna marcada como Restricted não pode ser lida mesmo que uma entrada de transformação esteja errada, e a verificação independente é a asserção de esquema em `verifyMasked`, que compara a cópia com o plano. Dentro desse grant, a carga decide coluna por coluna a partir do catálogo, e falha fechado.

```

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` faz duas verificações. Ela faz uma asserção sobre o esquema: o conjunto de colunas do destino deve ser igual ao plano, e nenhuma coluna `Restricted` ou descartada pode existir nele. Depois, amostra linhas e falha se uma coluna transformada ainda for igual ao seu valor de origem. A asserção de esquema pega uma coluna que chegou por um caminho que o plano não listava; a amostra pega uma transformação configurada de forma errada. Nenhuma das duas prova que a cópia não contém nada sensível, porque a amostra lê apenas algumas linhas.

O preço de falhar fechado é visível: quando uma pessoa desenvolvedora adiciona uma coluna e esquece o catálogo, a verificação da migração deveria tê-la barrado, e, se não barrou, a tabela de `analytics` carrega sem essa coluna e a equipe de dados nota uma lacuna. Aceitaria isso. Um relatório com uma coluna faltando é um ticket; uma cópia com uma coluna sem máscara é uma exposição que ninguém percebe.

Um pseudônimo com chave para joins merece uma decisão própria. Ele permite que os analistas acompanhem um mesmo cliente entre cargas sem saber quem é o cliente, e só funciona enquanto a chave permanece fora do alcance da equipe de dados e inalterada. Uma rotação da chave muda to-

dos os pseudônimos, e o histórico deixa de fazer join. Para muitas equipes, o `customer_id` opaco já cumpre essa função, e um segundo substituto acrescenta uma chave a proteger sem nenhum ganho.

Tokenização e criptografia na camada da aplicação

A **tokenização** substitui um valor por um substituto que só pode ser mapeado de volta por um serviço controlado. Não é mascaramento, porque o mapeamento existe, e em geral não é criptografia, porque o substituto normalmente não é calculado a partir do valor. Os esquemas que o calculam a partir do valor sob uma chave são criptografia com outro nome e exigem o mesmo cuidado com a chave. A Ledgerly já se beneficia disso em um ponto: `token_value` é um token do provedor, o provedor guarda o mapeamento e `payments-service` é o único chamador. O banco de dados da Ledgerly, portanto, nunca guarda o instrumento.

Construir um segundo serviço de tokenização para números de contas bancárias é um passo maior. Ele acrescenta ao caminho de pagamento um sistema que precisa estar disponível quando os pagamentos estão, exige uma lista de chamadores autorizados a mapear um token de volta, e cada mapeamento é um evento que alguém deveria poder revisar. Não o construiria no porte da Ledgerly. Restringir as colunas, restringir o único serviço que as lê e criptografar os valores daria a maior parte da proteção com menos coisas para manter em funcionamento. Se um parceiro regulado ou um provedor de pagamentos o exigir, a exigência decide.

A **criptografia na camada da aplicação** significa que o serviço criptografa um valor antes que o banco de dados o veja, de modo que papéis, réplicas, backups e dumps guardam todos texto cifrado. Essa é sua força distintiva: ela protege o valor nas cópias de dados que um grant não alcança (Capítulo 13). A criptografia feita pela camada de armazenamento, em disco ou em um snapshot, protege contra um volume perdido e não faz nada contra uma consulta, e as equipes muitas vezes a confundem com controle de acesso. A criptografia na camada da aplicação está mais próxima de uma fronteira de verdade, e é um compromisso mais pesado.

- **A chave decide quem pode ler.** Os dados são tão protegidos quanto a sua chave. Dê a permissão de descriptografar à `workload identity of payments-service` e a mais ninguém, de modo que a decisão seja

uma questão de quem pode descriptografar e por qual meio. Hierarquias de chaves, rotação e backup de chaves são uma disciplina à parte, e a documentação do serviço de chaves as cobre melhor do que este livro.

- **A coluna perde seus recursos de consulta.** Não é possível buscar, ordenar, filtrar por intervalo nem impor unicidade sobre texto cifrado da maneira usual, então escolha colunas que só são lidas de volta por identificador.
- **Um serviço que pode descriptografar sob demanda pode ser solicitado a isso.** A criptografia move a fronteira para o serviço, de modo que seus chamadores, sua taxa e suas entradas de auditoria passam a ser o controle, e a entrada de auditoria é escrita antes da descriptografia, como abaixo.

```
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)
```

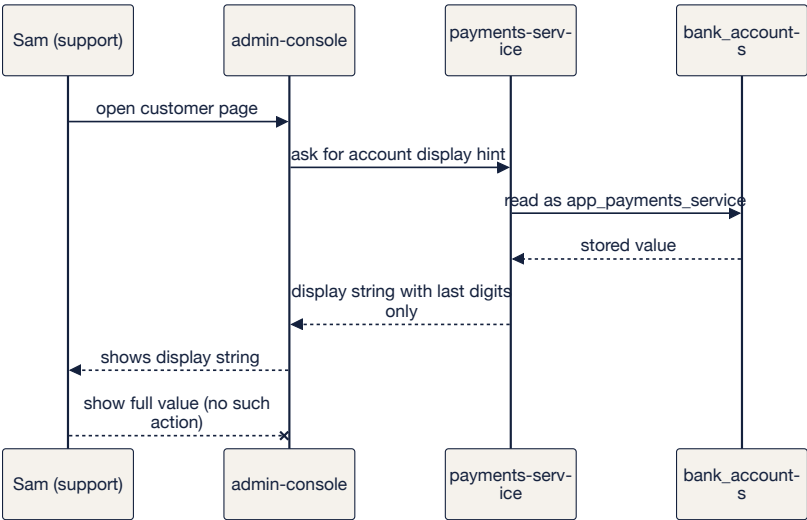
A matriz não muda se você criptografa. `human_breakglass` continua lendo as colunas Restricted, mas o que ele lê é texto cifrado, e descriptografar exige uma permissão de chave que o papel não possui. Decidiria com antecedência se um incidente que precisa do texto simples pode pedir acesso à chave, com qual aprovador, e registraria isso no runbook de break-glass do Capítulo 8. Deixar essa escolha para o incidente é como uma regra acaba sendo improvisada.

Trabalhar sem ver: o caso do suporte

De volta a Sam. O objetivo é que Sam consiga responder plenamente ao cliente sem jamais ver o valor completo. A maioria das perguntas do suporte acaba exigindo menos do que o valor.

O que o suporte precisa fazer	O que precisa ver	Como é fornecido
Dizer ao cliente qual conta está cadastrada	Os últimos dígitos	<code>payments-service</code> devolve um texto de exibição; ou a dica é armazenada quando a conta é gravada, como <code>last_four</code> é para os tokens
Verificar se a pessoa que escreve é o contato	Um sim ou não	Uma verificação estreita no serviço que já detém o grant, que compara o que o cliente disse com o valor armazenado e responde “confere” ou “não confere”
Consultar as faturas do cliente	Status, valores, datas de vencimento, somente deste cliente	<code>app_admin_console</code> lê as colunas <code>Internal de invoices</code> , e a regra de linha sobre o cliente do caso, do Capítulo 3, restringe as linhas
Corrigir um número de conta errado	Nada, e não à mão	O caminho da aplicação: <code>admin-console</code> pede a <code>payments-service</code> , uma segunda pessoa aprova e a mudança é registrada em <code>audit_events</code>

O diagrama mostra a primeira linha. Observe a última seta: uma ação que não existe é um controle mais forte do que uma que existe e é recusada.



Não construiria um botão de “revelar” no início. Um botão de revelar que exige um motivo e registra o clique parece razoável, e em um mês clicá-lo é um hábito e o seu log é algo que ninguém lê. Se surgir uma necessidade real, faça dela uma requisição com aprovador, como o Capítulo 8 faz para

qualquer leitura Confidential sem máscara, e conte com que frequência é usada. Uma funcionalidade usada todos os dias está dizendo que falta algo na forma de exibição.

Quem está de on-call segue a mesma regra sob outra pressão. Robin, depurando um lembrete que falhou, precisa saber que um endereço de destinatário foi rejeitado e raramente precisa do endereço. Quando Robin realmente precisa do `contact_email` sem máscara, isso é um `SELECT JIT` que nomeia a coluna, com um aprovador que lê a lista de colunas. As leituras de on-call nunca são aprovadas automaticamente: uma regra que pula as colunas Confidential aprova uma requisição que não concede nada, e uma regra que as inclui remove a única verificação que tem significado. O custo é uma espera em hora ruim, e a maneira de amortizá-lo é eliminar a necessidade do valor nos casos comuns, o que a próxima seção trata.

Dados sensíveis em logs e mensagens de erro

Os logs são a cópia de dados que ninguém projeta. São escritos por muitos componentes, lidos por muitas pessoas, mantidos por muito tempo e pesquisados com ferramentas que nunca fizeram parte do modelo de acesso. Quatro lugares concentram a maior parte do vazamento.

- **Logs da aplicação e mensagens de erro.** Uma mensagem construída a partir de um valor, como “não foi possível enviar para o endereço”, coloca o valor no log, no rastreador de erros e às vezes na tela de um cliente. Drivers e ORMs também podem registrar parâmetros vinculados quando o log de depuração está ligado.
- **Logs do banco de dados.** O log de instruções pode registrar literais, e um log de consultas lentas guarda o texto de uma consulta junto com os valores nela contidos.
- **Registros de instruções.** O access gateway registra cada instrução que uma pessoa envia (Capítulo 7), e uma pessoa que digita um valor em uma condição colocou esse valor no registro.
- **Tickets e chat.** Um resultado colado para explicar um problema é uma cópia com o público errado e sem data de término.

Aviso: Um log de instruções contém literais, de modo que o log de auditoria do Capítulo 14 e os registros de instruções do gateway são, eles próprios, armazenamentos de dados sensíveis. Registre as instruções como fingerprints com os literais removidos por redaction, classifique o armazenamento como no mínimo Confidential, restrinja quem o lê e dê a ele um período de retenção; caso contrário, o controle que vigia os dados se torna mais uma cópia deles. O Capítulo 14 trata do mecanismo.

Minha regra preferida para o logging da aplicação é uma allowlist, não um scrubber. Um scrubber procura padrões que se pareçam com valores sensíveis e os remove; ele pega os formatos que seu autor imaginou e deixa passar o resto. Uma allowlist nomeia os campos que podem aparecer e descarta todo o restante, de modo que um campo novo fica em silêncio até que alguém escolha registrá-lo.

```
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
```

As mensagens de erro seguem a mesma disciplina. Um erro carrega um código e identificadores, como um id de fatura, um id de cliente e um status de entrega, e não o valor que falhou. É assim também que o caso do lembrete de Robin deixa de precisar do endereço bruto: se o registro da falha contém um status como “destinatário rejeitado” e o id da fatura, o log responde à pergunta sem o endereço, e o registro do gateway sobre quem leu o endereço, se alguém leu, fica em um único lugar.

A redaction no pipeline de logs é uma rede de proteção, e deixará coisas passar. A verificação que não pularia é um canary nos testes: rodar a aplicação com um valor de teste de formato reconhecível em todo campo sensível e fazer o build falhar se ele aparecer em qualquer saída de log. Ela

testa a allowlist onde importa, e custa poucas linhas. A depuração fica mais lenta quando o log contém identificadores em vez de valores, e o `dropped_field_count` é para onde vão as mensagens do tipo “por que não consigo ver isto”, em vez de uma pessoa desenvolvedora reativar o logging de parâmetros em produção.

As decisões de colunas da Ledgerly

A tabela abaixo reúne o capítulo. É o registro de decisão das colunas sensíveis das oito tabelas, na forma em que uma revisão de acessos consegue lê-lo. Nessa coluna, “Retida” e “Sem acesso” significam ambas que o papel não tem grant na coluna, de modo que uma consulta que a nomeia é recusada. A coluna `human_readonly` indica o padrão de uma sessão; a coluna JIT indica se uma requisição pode nomear a coluna.

Coluna	Classe	<code>human_readonly</code>	SELECT JIT	Cópia de analytics
<code>bank_accounts.iban, account_holder_name</code>	Restric- ted	Sem acesso	Nunca	Descartada
<code>payment_tokens.token_value</code>	Res- tric- ted	Retida	Nunca	Descartada
<code>payment_tokens.provider, last_four, expires_on</code>	Con- fiden- tial	Legível	Desne- cessário	<code>provider</code> e <code>expires_on</code> mantidas, <code>last_four</code> descartada
<code>customers.contact_name, contact_email, contact_phone</code>	Con- fiden- tial	Mascarada	Nomeia a colu- na, sem máscara	Descartada
<code>customers.billing_address</code>	Con- fiden- tial	Com redaction	Nomeia a colu- na, sem máscara	Descartada
<code>payments.provider_reference</code>	Con- fiden- tial	Últimos quatro caracteres	Nomeia a colu- na, sem máscara	Descartada

Coluna	Classe	human_readonly	SELECT JIT	Cópia de analytics
invoice_lines.description	Confidencial	Retida	Nomeia a coluna, sem máscara	Descartada
amount_minor, currency, status, due_on, account_code	Internal	Legível onde a matriz concede	Desnecessário	Copiada

Duas entradas merecem um comentário. As colunas Restricted são descartadas da cópia de analytics e nunca são oferecidas a um papel humano que não seja `human_breakglass`; a única rota que as altera é o caminho da aplicação, com uma segunda pessoa aprovando (o Capítulo 9 explica como a política de execução encaminha as operações com gate para lá). O mesmo caminho cuida do vínculo de proprietário: `customer_id` em `bank_accounts` e `payment_tokens` fica dentro do teto JIT porque não é Restricted, mas mover uma linha para outro cliente muda de quem é a conta, então uma mudança de vínculo de proprietário passa por gate e é encaminhada ao caminho de produto, nunca feita como uma correção de dados comum. E, onde o motor não tem grants em nível de coluna, a mesma tabela continua valendo, com uma mudança: as colunas Restricted vivem nas tabelas de secrets, a coluna JIT diz “nível de tabela” e a tela de aprovação diz isso em palavras.

Trade-offs e custos

Classificar dá manutenção. Um catálogo que está correto no dia em que é escrito começa a envelhecer imediatamente. Alguém precisa ser responsável por cada decisão de classe, revisar as provisórias e responder à pergunta que uma coluna não classificada levanta. Se ninguém revisa o catálogo, ele se torna o tipo de registro que está completo e errado, o que é pior do que uma lacuna honesta.

Mascarar custa tempo de depuração. Robin não consegue ver o email bruto, então um bug que depende dos caracteres exatos dele precisa de uma requisição JIT e de um aprovador. Aceito esse custo pelo acesso que este capítulo fecha. Ele não é gratuito, e uma equipe que o paga toda noite começará a procurar um jeito de contorná-lo, então meça com que frequência a leitura sem máscara é solicitada e corrija a causa.

O mascaramento dinâmico depende do motor. O comportamento em torno de filtros, grants diretos e views difere, e uma máscara que você não testou no seu motor é uma esperança. A variante baseada em views acrescenta objetos que as migrações precisam manter em sincronia com as tabelas base.

Falhar fechado produz lacunas. A carga de analytics que recusa colunas não classificadas deixa a equipe de dados sem uma coluna até que alguém a classifique. Diga quem é responsável por essa espera e mantenha a revisão de classificação rápida o bastante para que a lacuna dure horas, não semanas.

A criptografia na camada da aplicação remove recursos e acrescenta uma dependência. O serviço de chaves fica no caminho de toda leitura da coluna, e a sua queda é uma queda de pagamentos. Como a entrada de auditoria vem primeiro, um log de auditoria inacessível também interrompe essas leituras, o que é um acoplamento deliberado e mais uma dependência de disponibilidade. A perda de recursos de consulta empurra a busca para outras colunas, e o backup da chave é mais uma coisa que precisa ser testada.

Alguns controles aqui podem virar teatro de segurança. Um botão de remover máscara que é sempre clicado, um catálogo que ninguém revisa, um scrubber de logs que ninguém testa e uma coluna mascarada que qualquer leitor pode sondar com filtros acrescentam atrito ou burocracia e não reduzem nada.

Quando não usar tudo isso. Um sistema pequeno, com um punhado de tabelas e sem dados pessoais de clientes, não precisa de um arquivo de catálogo, de uma carga mascarada nem de um serviço de chaves. Uma lista de uma página com as colunas sensíveis, grants em nível de coluna que as deixem de fora e uma allowlist de logs dão a maior parte do valor. Acrescente o mascaramento quando um segundo público precisar dos dados, acrescente a criptografia quando uma cópia em torno da qual você não consegue conceder grants se tornar uma preocupação, e registre por escrito qual evento faria você passar ao próximo passo.

Erros comuns

- **Tratar a criptografia em repouso como controle de acesso.** É tentador porque é uma única configuração e satisfaz um checklist. Ela protege contra um disco perdido e não faz nada contra uma consulta. Mantenha os grants e use a criptografia na camada da aplicação onde uma cópia precisar de proteção.
- **Mascarar no cliente e não no banco de dados.** Uma máscara na interface web é fácil de construir, e uma conexão direta, um export ou uma resposta de API a contornam. Mascare onde o dado sai do armazenamento.
- **Classificar tabelas e não colunas.** Um rótulo por tabela protege demais o flag de status e protege de menos o telefone. Classifique cada coluna e derive o rótulo da tabela a partir delas.
- **Definir Internal como padrão para colunas novas.** Internal é a classe de menor atrito, então se torna a classe de tudo o que não foi revisado. Faça de Confidential o padrão e libere por revisão.
- **Mascarar texto livre com padrões.** Um padrão encontra os números e emails que seu autor esperava e deixa passar um nome digitado em uma anotação. Retenha as colunas de texto livre dos leitores amplos.
- **Usar um hash sem chave como pseudônimo.** Ele parece irreversível. Para um valor com poucas possibilidades, pode ser revertido testando-as. Use um pseudônimo com chave, cuja chave fique fora do banco de dados, ou descarte a coluna.
- **Copiar produção para staging “e mascarar depois”.** A janela entre a cópia e a máscara é uma cópia de tudo. Mascare no caminho, como faz a carga de analytics, e trate o staging como uma das portas do Capítulo 13.
- **Adicionar uma coluna Restricted a uma requisição JIT para poupar um passo.** O teto a recusa por um motivo, e a tentação é maior à noite. Os valores Restricted só mudam pelo caminho da aplicação, com aprovação four-eyes, e em motores sem grants em nível de coluna ficam nas tabelas de secrets.
- **Aprovar automaticamente as leituras de on-call.** Isso remove a pausa em que uma pessoa verifica qual coluna Confidential está sendo exposta sem máscara. Mantenha o aprovador e reduza o número de vezes em que o valor é necessário.

- **Registrar corpos de requisição no nível de erro.** O corpo é a coisa mais útil para quem desenvolve e contém todos os campos que um cliente digitou. Registre identificadores e status, e deixe que uma allowlist decida o que mais aparece.

Pontos-chave

- Classifique toda coluna, trate o texto livre pelo pior caso e faça as colunas não classificadas assumirem Confidential por padrão, de modo que os dados novos comecem fechados e o trabalho de abri-los fique visível.
- Mantenha o catálogo de classificação no repositório que contém as migrações e bloqueie o merge que adiciona uma coluna sem classe, porque a verificação custa uma linha e a sua ausência é a falha.
- A minimização vem primeiro: o dado que nunca foi armazenado não precisa de máscara, e os dados pessoais mantidos fora das tabelas append-only não podem se tornar um problema de apagamento.
- O mascaramento dinâmico serve para `human_readonly` e protege contra exposição casual; o mascaramento estático serve para a cópia de analytics, onde o valor bruto pode ser totalmente descartado e a carga falha fechado diante de qualquer coisa não classificada.
- Dê às pessoas a forma do valor de que a tarefa precisa, como um texto de exibição ou um sim ou não, e mantenha as colunas Restricted fora de toda requisição JIT e de toda escrita em SQL bruto.
- Uma máscara no stream de resultados é uma segunda camada atrás do grant, nunca um substituto dele, e um papel de carga que não consegue ler colunas Restricted faz um erro de catálogo falhar fechado.
- Logs, mensagens de erro e registros de instruções são cópias de dados: registre identificadores por meio de uma allowlist, mantenha valores fora das mensagens e classifique o próprio log de auditoria.

Os controles deste capítulo protegem os dados enquanto estão no banco de dados e enquanto uma consulta os devolve. O próximo capítulo acompanha os dados para fora, em backups, dumps, exports e cópias de staging, onde nenhum desses papéis se aplica.

Capítulo 13 — Backups, exports e outras portas laterais

Imagine uma sexta-feira à tarde na Ledgerly. Robin está atrás de um bug nos lembretes de faturas que só aparece com dados realistas: faturas com datas de vencimento incomuns, clientes com vários contatos, descrições de linhas com texto livre que ninguém previu. O staging tem uma dúzia de clientes feitos à mão, e o bug não aparece em nenhum deles. Alguém se lembra de que um dump antigo de produção está em um armazenamento compartilhado, e Alex, que cuida dos bancos de dados, o restaura no staging para que a equipe toda possa olhar.

O bug é encontrado em menos de uma hora. O dump continua na pasta, e o banco restaurado continua no staging. O staging não tem access gateway, porque não é produção. O login dele é compartilhado, todo engenheiro consegue alcançá-lo, jobs de CI rodam contra ele, e um prestador de serviço que entrou no mês passado tem uma conta. O `invoice-worker` do staging usa as mesmas configurações de email de saída que o de produção, e o job de lembretes está ligado.

Ninguém decidiu expor dados de clientes. Cada passo foi uma resposta sensata ao problema de uma tarde, e os controles construídos nos capítulos anteriores protegem um banco de dados que já não guarda a única cópia do próprio conteúdo.

Este capítulo acrescenta o último passo do estado-alvo antes da Parte V: backups, exports e cópias de staging são tratados como portas com responsável e expiração. A pergunta é esta: quando os dados saem do primary de propósito, para recuperação, para um fornecedor, para um teste ou para uma pessoa que precisa olhar, como manter os controles junto com os dados e não com o banco de dados de onde eles vieram?

No modelo de cinco partes (**Identidade, Grant, Caminho, Execução, Registro**), uma cópia é um lugar onde o Grant e o Caminho já não se aplicam, porque os dados saíram por uma rota que não carrega nenhum dos

dois. O trabalho do capítulo é fazer a própria cópia carregar um Grant: uma lista de leitores, um responsável e uma data de término, com um Registro de quem a criou e de quem a removeu.

O problema e por que ele é difícil

O Capítulo 6 definiu uma **porta lateral** como uma cópia de dados ou um caminho de export que contorna os controles de acesso do primary. As réplicas e a cópia de analytics são portas que você constrói deliberadamente e sabe nomear. Este capítulo trata do resto: backups e snapshots, dumps, arquivos CSV, bancos de staging, alguns milhares de linhas em um laptop, o cache de um dashboard, um arquivo na caixa de entrada de um fornecedor. A maioria foi criada por um bom motivo. Poucas têm responsável.

A dificuldade central é que os controles de acesso vivem em um sistema, não nos dados. O primary tem um access gateway, um login por sessão, grants em nível de coluna, timeouts e um log de auditoria. Um dump do primary não tem nada disso, porque um dump é um arquivo. Quem consegue ler o arquivo tem tudo o que há nele, inclusive as colunas que o esquema de papéis retém de todo papel humano exceto `human_breakglass`. Um backup que contém as colunas Restricted é, para quem consegue lê-lo, um grant permanente sobre essas colunas que nenhum gateway emitiu e nenhuma revisão lista.

A resposta óbvia é uma política: não copie dados de produção. Ela falha por um motivo simples. Copiar é como o trabalho é feito. A recuperação precisa de uma cópia por definição. Um fornecedor precisa de arquivos. Um bug que depende do formato real dos dados precisa de algo que pareça real. Uma política que proíbe tudo isso é contornada, e as cópias que resultam são as não gerenciadas. A melhor pergunta é quais cópias são necessárias, quem é responsável por cada uma, quem pode lê-la e quando ela termina.

Uma segunda dificuldade é que essas portas são criadas por equipes diferentes por motivos diferentes, então nenhum responsável único enxerga todas. A equipe de plataforma é responsável pelos backups, a equipe de dados pela cópia de analytics, a equipe de backend pelo staging, o financeiro pela relação com o fornecedor, e cada engenheiro pelo conteúdo do

próprio laptop. Um inventário é o primeiro controle, e ele se encontra em buckets de armazenamento, agendas de jobs e configurações, não em um documento de política.

Um mapa das portas laterais

Antes dos controles, um mapa. A tabela lista as portas que este capítulo cobre, o que cada uma guarda no caso da Ledgerly e quem a alcança por padrão se ninguém projetou o contrário.

Porta	O que guarda	Quem alcança por padrão	Primeiro controle
Backups e snapshots de volume	Tudo, inclusive as colunas Restricted e as tabelas de secrets	Qualquer pessoa com acesso ao armazenamento ou à infraestrutura	Poderes separados para criar, restaurar e excluir; criptografia com chave própria
Dumps em armazenamento compartilhado	Tudo, como um arquivo legível	Qualquer pessoa que consiga abrir a pasta	Encontrar, atribuir um responsável, excluir ou mover para as regras de backup
Exports humanos (CSV, tabelas coladas)	O que a consulta retornou	A pessoa e depois quem ela enviar	Aprovação por faixas, expiração, nenhum dado Restricted
Bancos de staging e de teste	Uma cópia de produção, ou algo parecido	Todo engenheiro, CI, prestadores de serviço	Dados mascarados ou sintéticos, mensagens de saída desligadas
Laptops	Conjuntos de resultados, bancos locais, arquivos salvos	O usuário do laptop e seus backups	Eliminar a necessidade; amostras mascaradas no lugar
Ferramentas de BI	Extratos em cache, relatórios agendados, dashboards compartilhados	Todos com quem a ferramenta é compartilhada	Ler somente a cópia de analytics, com identidade por pessoa quando possível
Arquivos de fornecedores	Exports de faturas e do ledger	A equipe e os sistemas do fornecedor	Papel estreito, lista de colunas, retenção, expiração do arquivo

Toda linha tem o mesmo tipo de problema: uma cópia de dados com os controles de um sistema de arquivos ou de um produto SaaS, e uma lista de leitores que ninguém escreveu. O registro do Capítulo 6 já existe por

esse motivo, e o estenderia a todas as linhas acima. Uma porta que não está no registro não é uma porta que alguém vá 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
```

Os campos novos são `storage` e `deleted_by`; `kind` ganha os valores `backup`, `laptop`, `bi_extract` e `vendor_file`, e `contains` registra a classe de dados mais alta presente. O primeiro campo novo diz onde olhar quando alguém pergunta o que consegue ler a cópia, porque a resposta costuma ser uma permissão de armazenamento e não um papel do banco de dados. O segundo fecha o ciclo: uma data em `ends` sobre a qual ninguém age é um desejo, e o registro de quem excluiu a cópia é a evidência de que a data significava alguma coisa.

Backups e snapshots

Um backup é a única porta lateral contra a qual ninguém argumenta, e esse consenso é parte do motivo pelo qual ela tantas vezes fica sem controle. Ele precisa guardar tudo, porque sua função é restaurar tudo. Esse é todo o projeto, e é também toda a exposição.

O que um backup guarda

Os backups vêm em alguns tipos. Um **backup físico** ou snapshot de volume copia os arquivos do banco de dados, muitas vezes com um arquivo contínuo de mudanças para que o banco possa ser restaurado a um ponto no tempo. Um **dump lógico** lê os dados por meio do banco de dados e os grava como instruções ou linhas. Ambos guardam todas as tabelas. O tipo

físico também guarda as definições de papéis do cluster e o material de credencial que o motor armazena para eles, e algumas ferramentas lógicas também, dependendo de como são usadas.

Duas consequências decorrem disso, e nenhuma é visível em um diagrama do primary. A primeira é que as colunas Restricted estão no backup, e as tabelas de secrets também, nos motores que as usam (`bank_account_secrets` e `payment_token_values`, do Capítulo 3). A regra do esquema de papéis, segundo a qual somente os papéis da aplicação e `human_breakglass` alcançam esses valores, descreve o banco de dados. Ela não diz nada sobre o arquivo.

A segunda é que o backup guarda o estado do dia em que foi feito. Um backup de antes da aposentadoria do login de admin compartilhado ainda contém esse login. Ele guarda as associações de papéis de pessoas que depois mudaram de equipe. A rotação e a revogação acontecem no sistema vivo e não alcançam retroativamente os arquivos que já existem. Quando alguém estreita um papel no primary, nada acontece com a restauração do ano passado: o primary melhora, e as cópias mantêm o estado do dia em que foram feitas.

Restaurar é ler

Poria uma frase no topo do projeto de backup: o poder de restaurar é o poder de ler tudo. Quem consegue restaurar um backup em um lugar que controla pode então entrar ali como proprietário e ler todas as colunas, e nenhum gateway, grant de coluna ou regra de mascaramento de produção se aplica, porque a leitura não acontece em produção.

Isso muda a quem pertence o controle. O acesso a backups costuma ser concedido por papéis de infraestrutura: alguém pode ler o bucket, montar o snapshot ou rodar a ferramenta de restauração. Esses papéis raramente são revisados com o mesmo cuidado que `human_readonly`, embora estejam acima dele. A pergunta para uma revisão de acessos, portanto, não é apenas “quem pode consultar o primary”, mas “quem pode restaurar, e onde”.

Separaria três poderes e os daria a principais diferentes:

Poder	O que significa	Quem o detém
Criar	Criar backups e gravá-los no armazenamento	A identidade própria do job de backup, somente com escrita no armazenamento
Restaurar	Ler um backup e colocá-lo no ar em algum lugar	A equipe de plataforma, por requisição, com uma segunda pessoa aprovando (four-eyes)
Excluir	Remover ou expirar backups	O processo de retenção; pessoas somente por meio de uma requisição revisada

O sentido da divisão é que o job que grava os backups não consegue lê-los de volta, e ninguém pode ao mesmo tempo restaurar um backup e excluir os pontos de recuperação, de modo que uma restauração não possa ser seguida da remoção daquilo de que foi restaurada. Onde o armazenamento oferece retenção write-once por um período, a usaria pelo mesmo motivo, a um custo: você não consegue excluir um backup feito por engano, como um tirado de um banco de dados que por pouco tempo guardou algo que não devia, até que o período termine. Decida com antecedência como esse caso será tratado.

Criptografia e de quem é a chave

Os backups devem ser criptografados em repouso, e a pergunta interessante é quem detém a chave. Se o mesmo administrador consegue ler o bucket e usar a chave, a criptografia protege contra um disco perdido e nada mais. Se a chave é mantida por um principal diferente dos administradores do armazenamento, e a restauração exige ambos, então a equipe de armazenamento sozinha não consegue ler os dados e quem detém a chave sozinho não tem nada para ler.

Essa separação custa algo específico. A restauração passa a exigir duas partes, e um backup cuja chave se perdeu deixa de ser um backup. Um caminho de recuperação da chave precisa existir e ser testado, o que é mais um procedimento para a equipe de plataforma manter. O Capítulo 12 trata da criptografia na camada da aplicação e dos limites do gerenciamento de chaves; o ponto mais estreito aqui é que a criptografia do backup é um gate sobre a restauração, e um gate só vale a pena se for um gate diferente do que protege o armazenamento.

A criptografia não ajuda contra a pessoa que tem permissão para restaurar. Ela protege o arquivo em trânsito e em repouso de pessoas e sistemas que não fazem parte do caminho de restauração. Diria isso com clareza no documento de projeto, porque “os backups são criptografados” costuma ser ouvido como “os backups são seguros”, e são afirmações diferentes.

Uma requisição de restauração

A restauração é o momento em que o backup se torna um banco de dados vivo, então o controle pertence a ela. O bloco abaixo mostra uma requisição de restauração com a ordem das operações escrita por extenso. O armazenamento de backups, o registro e o log de auditoria são sistemas separados, de modo que nenhuma transação os cobre.

```

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)

```

Se o processo parar depois de `registerCopy` e antes de a restauração começar, o registro contém uma entrada para uma cópia que não existe, o que uma verificação semanal encontra e remove. Se parar depois que a restauração começou, a cópia existe e está inalcançável, e a exclusão agendada a remove; se até o agendamento falhou, o sweeper que age sobre `ends` no registro a encontra. A ordem inversa deixaria uma cópia que exis-

te e não está listada, e essa é a falha que importa. O custo do projeto é o atraso: uma aprovação e um destino isolado antes que alguém veja uma linha. Durante um incidente, esse atraso é real, e é por isso que o caminho de restauração deve ser ensaiado, com a aprovação combinada com antecedência, como o Capítulo 15 discute para os incidentes. Quando o objetivo é recuperar algumas linhas e não um banco de dados inteiro, a restauração vai para o mesmo destino isolado, as linhas são retiradas dele e voltam para produção como uma correção de dados sob as regras do Capítulo 10.

Uma cópia restaurada começa fechada

Um banco de dados restaurado traz de volta os papéis, as associações e o material de credencial do dia do backup. Trataria o banco restaurado como hostil até que a definição de acesso tenha sido aplicada novamente. Na prática, isso significa um destino de restauração sem rota de rede a partir de serviços e sem logins humanos, e um primeiro passo que desabilita todo login que o backup trouxe de volta, o superusuário e `ledgerly_access_admin` inclusive, porque cada um volta com a credencial que tinha no dia do backup. Depois a credencial do superusuário é rotacionada, e só então a definição de acesso do Capítulo 3 é aplicada: papéis, associações e grants como o código os descreve agora. A cópia se torna alcançável depois disso, e não antes.

Esse passo exige um administrador de acesso, e ele precisa ser um que pertença ao destino. O backup já guarda um login chamado `ledgerly_access_admin`, que voltou com a credencial do dia em que o backup foi feito. A restauração o substitui: o login que voltou é removido e criado de novo com o mesmo nome e uma nova credencial, feita para a restauração e removida junto com ela. A credencial atual de produção para esse papel nunca é copiada para o destino, e a que está no backup deixa de existir antes que algo possa se conectar, porque quem detém qualquer uma das duas pode conceder qualquer coisa dentro dos papéis que administra, e um simulado de restauração é exatamente o lugar onde as credenciais acabam coladas em um ambiente de teste. Se o administrador da própria cópia restaurada for comprometido, o raio de impacto é essa cópia.

Somente depois disso a cópia restaurada recebe leitores, e os leitores são as pessoas indicadas na requisição de restauração. Uma restauração para um simulado não precisa de nenhum leitor humano: o simulado verifica se os dados sobem e se as contagens batem, e depois a cópia é excluída.

Retenção e exclusão

A retenção é uma troca entre dois custos. Mantenha os backups por mais tempo e você consegue se recuperar de problemas percebidos tarde, ao preço de uma pilha maior de dados que guarda tudo, inclusive dados de clientes que o primary excluiu há muito tempo. Mantenha-os por menos tempo e você guarda menos, e alguma corrupção será mais antiga que o seu backup mais antigo. Não sei qual é o período certo para a sua empresa. Ele depende de quanto tempo um problema silencioso costuma levar para ser percebido, de quanto custaria um mês perdido de dados e do que o jurídico e a conformidade dizem sobre dados excluídos que continuam vivendo nos backups. Regimes de privacidade como GDPR ou KVKK tendem a perguntar o que acontece com os dados de uma pessoa depois que são excluídos do sistema vivo, e a interpretação é deles, não deste livro. O que a engenharia pode fornecer é evidência, não conformidade: um registro e um registro de exclusão que mostram o que o sistema faz.

Um projeto que vale a pena discutir com eles: os backups expiram em um cronograma, e um procedimento de restauração reaplica as exclusões registradas desde que o backup foi feito, de modo que uma cópia restaurada não traga de volta dados removidos de propósito. Isso acrescenta um passo a toda restauração e exige que as exclusões sejam registradas em um lugar que uma restauração possa consultar, o que é mais um motivo para manter as requisições de exclusão em um registro durável. O passo roda depois que a definição de acesso é aplicada e antes que a cópia seja aberta.

```
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
```

Simulados criam cópias

Um backup que nunca foi restaurado não demonstrou funcionar. Os simulados de restauração são como você descobre que os backups funcionam, e cada simulado cria um banco de dados contendo dados Restricted. Executaria os simulados pelo mesmo caminho de requisição de uma restauração real, com o mesmo destino isolado, a mesma entrada no registro e uma data de término dentro do dia. O simulado é também o ensaio do incidente: se ele precisa de um atalho para terminar, o atalho será tomado às três da manhã.

Aviso: Um backup ou dump que contém colunas Restricted e está em um armazenamento legível por um grupo amplo é a mesma exposição de um grant permanente sobre essas colunas, sem gateway e sem log. Se você encontrar um, restrinja primeiro a permissão do armazenamento e converse sobre retenção depois.

Dumps, exports e arquivos CSV

Os backups são a porta com propósito e com equipe. Os dumps e exports são as portas geralmente abertas por uma única pessoa sob pressão de tempo, e vêm em dois tipos que devem ser tratados de forma diferente: os exports que acontecem de forma agendada e os que uma pessoa faz.

Dumps feitos à mão

A pasta compartilhada da Ledgerly guarda alguns dumps antigos. Cada um foi feito porque alguém precisou de uma cópia por uma hora, e a cópia ficou. O dump de uma pessoa é um backup sem retenção, sem política de criptografia e sem responsável. É também, nos motores em que um dump lê através do banco de dados, evidência de que alguém tinha um login capaz de ler todas as tabelas.

Faria três coisas. Primeiro, tirar o dump do conjunto de ferramentas normais: o access gateway não oferece um dump do banco inteiro, os papéis humanos não conseguem ler as tabelas de que um dump precisaria, e `human_breakglass` é o único papel humano que poderia, com o conjunto de logging de break-glass do Capítulo 8. Segundo, encontrar os dumps existentes e, antes de excluir qualquer um, verificar se algum é o único backup de alguém. Isso acontece: um dump se torna o backup de uma tabela que ninguém mais copiou. Construa primeiro os backups de verdade e depois exclua os dumps. Terceiro, dar à pessoa que precisava do dump uma alternativa que cumpra a função, que em geral é a cópia de analytics mascarada ou um conjunto de dados de teste (veja a seção sobre dados de teste abaixo).

Exports humanos

Os exports feitos por pessoas são os mais comuns e os mais difíceis de proibir. O suporte precisa de uma lista das faturas de um cliente. O financeiro precisa de um arquivo com os pagamentos do mês passado. A equipe de dados precisa de uma amostra. Separaria esses exports em faixas pela classe de dados mais alta presente no arquivo, porque uma única regra que sirva a todos é frouxa demais para os perigosos ou pesada demais para os inofensivos.

Faixa	Conteúdo	Controle
Somente Internal	Agregados, contagens, detalhes por status, sem conteúdo pessoal	Autoatendimento, registrado, sem aprovação
Contém Confi-dencial	Linhas com contato de cliente ou texto livre	Ticket com motivo, um aprovador, uma data de término para o arquivo; as linhas vêm de uma sessão de <code>SELECT JIT</code> que nomeia as colunas (Capítulo 8), de modo que o aprovador lê a lista de colunas que o arquivo conterá

Faixa	Conteúdo	Controle
Contém Restric-ted	Valores de conta bancária ou de token	Não é exportado; a resposta é uma forma de exibição ou um caminho da aplicação (Capítulo 12)

A faixa é calculada no gateway, não declarada por quem solicita, e falha fechado: é a classe mais alta do catálogo (Capítulo 12) entre as colunas de origem que a consulta lê, não entre os nomes das colunas do resultado, porque um alias ou uma expressão como `lower(contact_email) AS x` esconde a origem. Uma coluna cuja origem não pode ser resolvida, por meio de uma view, de uma CTE ou de uma função, conta como a faixa exportável mais alta, e onde nem isso puder ser decidido o export é recusado. O Capítulo 9 trata dos limites da análise estática. Essas faixas são uma ilustração de uma forma. O que importa é que a aprovação esteja ligada ao risco. Uma aprovação concedida em segundos, sempre, para tudo, é o teatro que o Capítulo 8 descreve. A faixa que exige um aprovador humano deve ser pequena o bastante para que o aprovador leia a requisição, e mediria isso pelo fato de alguém dizer não alguma vez.

A data de término é a parte que as equipes pulam. Um export sem data de término é uma cópia que sobrevive à sua pergunta. O ticket que aprova o arquivo carrega uma data, o arquivo vai para um lugar com responsável e um sweeper o remove. A regra do Capítulo 10 para o trabalho ad hoc cobre o caso mais leve: cole identificadores e contagens, não linhas com conteúdo pessoal. O Capítulo 7 trata do que o gateway pode fazer diante de um conjunto de resultados em massa, que é limitá-lo e torná-lo visível, não impedi-lo.

O export para o fornecedor

O fornecedor externo de contabilidade nunca se conecta a um banco de dados da Ledgerly. Ele recebe arquivos, o que faz da etapa de export uma porta que você pode projetar por completo, porque controla as duas pontas do caminho até a caixa de entrada do fornecedor. No estado inicial, a etapa de export do `reporting-job` grava esses arquivos com o login de admin compartilhado, de modo que o arquivo pode conter qualquer coisa que o papel de admin consiga ler. No estado-alvo, ela usa `app_vendedor_export`, e isso muda o que o arquivo pode conter, porque um arquivo não pode conter mais do que o login que o gravou conseguia ler.

```
// 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;
```

O grant é o controle, não a consulta. Um erro na consulta de export pode deixar linhas de fora ou repeti-las, e não pode acrescentar `contact_email`, porque o papel não consegue lê-lo. `customer_id` é um identificador opaco e o fornecedor não tem tabela com a qual fazer join. Se o trabalho do fornecedor precisar de nomes de clientes, essa é uma decisão que alguém toma e aprova, e não algo que chega porque uma consulta foi ampliada.

A etapa de export se conecta a `ledgerly_replica_2`, como diz a política de conexão do Capítulo 6, e grava um arquivo. O arquivo então vai ao fornecedor pelo canal combinado com ele. Os passos envolvem três sistemas separados, então o bloco abaixo mostra a ordem e o que cada falha deixa.

```
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)
```

Aqui `audit0rQueue` acrescenta o registro ao log de auditoria e, se o log estiver inacessível, o enfileira para retry e notifica a equipe de plataforma, como no Capítulo 7. Uma falha do log se torna uma lacuna visível e repetida e não uma lacuna silenciosa, e nenhuma transação abrange o log e o canal do fornecedor.

O manifesto é a parte que as pessoas deixam de fora. Ele permite que os dois lados verifiquem que o arquivo é o que se pretendia e dá à revisão de acessos algo para ler: tantas linhas, estas colunas, este período. Se uma coluna aparece em um manifesto sem ter estado no do mês passado, isso é uma mudança que alguém deveria ter aprovado.

Dois limites permanecem. Quando o arquivo chega ao fornecedor, a retenção dele é uma questão de contrato e de confiança: você pode pedir ao fornecedor que confirme a exclusão e não consegue verificá-la. E o checksum mostra que o arquivo não foi alterado em trânsito, não que a equipe do fornecedor seja a única a lê-lo. Para esses dois pontos, colocaria um prazo de retenção no contrato e uma linha na revisão de acessos que pergunte quando o fornecedor confirmou a exclusão pela última vez.

Dados de teste: staging, cópias mascaradas e dados sintéticos

O staging é onde o cenário da Ledgerly começou, e é a porta mais frequentemente aberta por gentileza. Os engenheiros querem dados realistas, e a produção é a única fonte de dados com os formatos que os clientes reais criam. O custo é que o staging tem o modelo de acesso de um sistema de teste com os dados de um de produção.

Três tipos de dados fora de produção

Tipo	Realismo	Exposição	Custo
Cópia de produção	O mais alto	A mesma de produção, com controles mais fracos	Barata de fazer, cara de manter
Cópia mascarada	Alto, com alguns formatos perdidos	Menor, não zero	Um pipeline de mascaramento para construir e manter (Capítulo 12)

Tipo	Realismo	Exposição	Custo
Dados sintéticos	Menor, limitado ao que foi modelado	A menor	Um gerador para manter, e bugs que exigem formatos não modelados

Não copiaria dados de produção para o staging sem máscara, e o motivo não é abstrato. O staging costuma ter mais leitores que a produção, um login compartilhado, um CI que armazena logs e integrações configuradas para se comportar como as de produção. Ele combina o maior público com os controles de identidade mais fracos e guarda os dados mais sensíveis. Se os dados precisam ser reais, os controles precisam ser tão fortes quanto os de produção, e nesse ponto o staging é a produção com outro nome.

O que uma cópia mascarada continua sendo

O mascaramento, tratado no Capítulo 12, substitui ou obscurece valores mantendo o formato utilizável. Uma cópia mascarada é muito melhor que uma bruta, e continua sendo uma cópia de dados, por motivos que vale a pena declarar para que ninguém trate “mascarado” como sinônimo de “seguro para distribuir”.

Primeiro, o mascaramento cobre as colunas que você listou. Um texto livre como `invoice_lines.description` pode conter dados pessoais que a classificação nunca previu, e é por isso que o livro o classifica pelo pior caso, e uma máscara que o pula o deixa na cópia. Segundo, as máscaras que preservam o formato mantêm distinguíveis as linhas que restam: um valor de fatura incomum, uma data incomum e um cliente pequeno podem juntos apontar para uma única empresa. Os valores foram mascarados, e a combinação ainda carrega informação. Terceiro, o mascaramento consistente, em que a mesma entrada recebe a mesma saída para que os joins continuem funcionando, também permite que a mesma pessoa seja reconhecida entre cópias e tabelas.

Nada disso significa que o mascaramento não valha a pena. Significa que uma cópia mascarada é tratada como Confidential até que alguém revise o que sobrou, que o seu público é menor que “toda a engenharia” e que ela tem uma data de término como qualquer outra cópia; a cópia contínua de analytics é a exceção, e é revisada na sua data de `review_due`. A palavra para isso é **reidentificação**: a capacidade de descobrir de quem trata um

registro a partir de campos que foram deixados. Descreveria o risco à equipe nesses termos e não na linguagem dos ataques. A resposta de projeto é um público menor e uma expiração, não uma lista mais longa de campos mascarados.

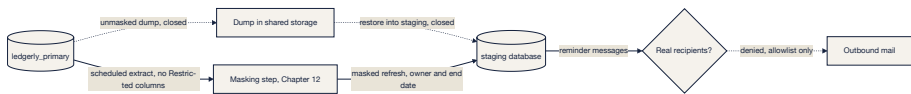
Dados sintéticos

Os dados sintéticos não contêm nenhuma pessoa real, então carregam a menor exposição. O limite deles é que contêm apenas os formatos que alguém pensou em modelar, e os bugs que exigem dados reais são os que ninguém modelou. Um gerador que cria clientes arrumados não cria o cliente com seis contatos e uma descrição em dois idiomas.

Faria o conjunto sintético crescer a partir dos bugs. Cada vez que um problema exige dados com o formato de produção para ser reproduzido, o formato que o causou, e não os valores, vira uma fixture: um cliente com vários contatos, uma linha de fatura com descrição vazia, uma data de vencimento no fim do mês. Ao longo de um ano, o conjunto deixa de ser arrumado e passa a ser o tipo útil de estranho. Isso custa a um engenheiro uma hora depois de cada bug desses, e elimina a necessidade da próxima cópia de produção.

O staging da Ledgerly, feito

Volte à sexta-feira de Robin. O caminho que abriu a porta foi um dump do armazenamento compartilhado para o staging. O caminho que construiria o substituiu por uma atualização que passa pela etapa de mascaramento e chega a um banco de dados de vida curta e com responsável nomeado. No diagrama, observe os caminhos tracejados: são os que o projeto fecha.



A metade direita do projeto é fácil de esquecer. Um ambiente de staging guarda dados e também age sobre eles. O `invoice-worker` do staging lê clientes e envia lembretes, e, se suas configurações de saída forem iguais às de produção, ele os envia para os endereços de contato que estão nos dados. Uma cópia mascarada não tem endereços reais, o que resolve isso por acaso. Uma cópia bruta transforma uma execução de teste em mensa-

gens para clientes reais. Faria o caminho de saída de todo ambiente fora de produção negar por padrão e permitir apenas uma allowlist de endereços internos de teste, independentemente do que os dados contenham.

Os tokens de pagamento seguem uma regra relacionada. Um token representa um instrumento de pagamento, e se ele funciona fora do sistema que o emitiu depende do provedor, o que não tenho como julgar para o seu. O padrão é que `payment_tokens.token_value`, `bank_accounts.iban` e `bank_accounts.account_holder_name` nunca estejam presentes em uma cópia fora de produção, mascarada ou não (os backups são a exceção, porque precisam guardar tudo), e o staging usa os tokens de teste do provedor.

Nota: Trate a atualização do staging como um pipeline com um caminho de falha, não como algo pontual. Se a etapa de mascaramento falhar, a atualização para e o staging mantém o conteúdo anterior. Uma atualização que recorre a uma fonte sem máscara por uma execução recria o dump na pasta compartilhada com passos extras.

Ferramentas de BI, laptops e fornecedores

Mais três portas merecem um tratamento breve, porque cada uma tira dados do banco de dados por uma rota que parece uma funcionalidade.

Ferramentas de BI. Uma ferramenta de dashboard se conecta com um único login, e todos com acesso ao dashboard veem o que esse login consegue ler. Isso inverte o modelo de identidade do Capítulo 3: o banco de dados vê um principal, e, digamos, cem pessoas veem seus dados. Dois controles ajudam. Aponte a ferramenta para `ledgerly_analytics` e não para uma réplica, de modo que ela leia dados mascarados com um papel próprio da ferramenta, um papel `app_` como qualquer outro workload (`human_analytics_reader` é o papel da equipe de dados para pessoas), em vez de papéis de produção. E, onde a ferramenta oferecer suporte, faça-a repassar a identidade de cada pessoa ao banco de dados, de modo que as regras de linha e de coluna se apliquem por pessoa. A ferramenta também mantém extratos, caches e relatórios por email agendados próprio-

os, que são cópias de dados e entram no registro. Um dashboard é uma porta conveniente, e um extrato em cache é a mesma porta sem os controles.

Laptops. O gateway remove credenciais salvas e define padrões somente leitura, como descreve o Capítulo 7, e não consegue controlar o que uma pessoa faz com as linhas que chegaram à tela. O controle prático é reduzir os motivos para manter uma cópia local: uma amostra mascarada, um banco de staging que se comporte como a produção e fixtures para os formatos que causam bugs. Uma equipe não consegue provar que nenhum laptop guarda dados de produção, e não afirmaria isso. Uma linha de base como a criptografia de disco completa e uma forma de apagar um dispositivo perdido reduz o que um laptop perdido custa, e é uma linha de base, não uma resposta.

Outros fornecedores. A contabilidade é o que o livro acompanha, e outros recebem dados por efeito colateral: um rastreador de erros que captura o corpo de uma requisição, um suporte que recebe screenshots colados, uma plataforma de logs que armazena instruções. Cada fornecedor que guarda uma cópia recebe uma linha no registro com o que contém e quando o contrato termina. O Capítulo 12 trata de manter valores sensíveis fora de logs e erros, que é a forma mais barata de mantê-los fora do armazenamento de um fornecedor.

Fechando as portas na Ledgerly

Considere como Alex levaria a Ledgerly de cópias espalhadas a um registro com responsáveis. Como no caminho de leitura do Capítulo 6, a ordem importa, e a faria em seis passos.

1. **Inventariar as cópias.** Liste backups, snapshots, dumps, exports, staging, extratos de BI e arquivos de fornecedores, com armazenamento, leitores e um responsável. Espere encontrar cópias de que ninguém se lembra; a primeira rodada do registro é sobretudo descoberta.
2. **Restringir primeiro o armazenamento.** Para tudo o que contém colunas Restricted, estreite quem pode ler o armazenamento antes de decidir o que excluir. Isso é rápido e reversível, e fecha a exposição mais ampla enquanto o resto é organizado.

3. **Construir os poderes de backup.** Dê ao job de backup uma identidade somente de escrita, coloque a restauração atrás de uma requisição com uma segunda pessoa aprovando, mantenha a chave de criptografia separada dos administradores do armazenamento e execute um simulado pelo caminho da requisição.
4. **Aposentar os dumps.** Quando os backups de verdade existirem e uma restauração tiver funcionado, exclua os dumps antigos e registre quem excluiu cada um. A restauração noturna sem máscara que alimenta `ledgerly_analytics` termina quando a carga mascarada do Capítulo 12, sob `app_analytics_load`, passa a rodar no lugar dela; até lá, é a maior porta aberta e recebe uma data de término.
5. **Mover o staging para dados mascarados ou sintéticos,** com mensagens de saída desligadas por padrão, e remover o login compartilhado do staging.
6. **Colocar o export do fornecedor em um login próprio.** Mova a etapa de export para `app_vendor_export`, acrescente o manifesto e defina uma vida útil para o arquivo.

Depois, ligue o registro a algo que acontece de qualquer modo. A revisão de acessos do Capítulo 14 pergunta quais cópias existem e quem pode ler cada uma, criar uma cópia exige uma linha no registro e um sweeper remove tudo o que passou de seu `ends` depois de um período de carência e de uma notificação ao responsável. As requisições de restauração e os exports também devem escrever no log de auditoria, que é o assunto do Capítulo 14. Os jobs de restauração e de export seguem a mesma disciplina de execução do runner que carrega as escritas aprovadas: rodam sob identidade própria, a partir de uma requisição aprovada, com o registro escrito primeiro, exceto que um export da faixa Internal mantém a regra de auto-atendimento da tabela de faixas acima (o Capítulo 9 descreve essa disciplina).

Trade-offs e custos

O primeiro custo é a velocidade durante um incidente. Uma restauração que precisa de um aprovador e de um destino isolado é mais lenta do que uma feita por quem tem a permissão do bucket, e o tempo que ela acrescenta recai sobre quem está de on-call no pior momento. Ensaie o cami-

nho e combine a aprovação com antecedência (o Capítulo 15 trata do acesso durante incidentes), para que o controle seja um procedimento que as pessoas conhecem e não um obstáculo que descobrem.

O segundo custo é a custódia da chave. Separar a chave do backup do armazenamento significa que uma chave perdida é um backup perdido, e o caminho de recuperação da chave é mais um procedimento que precisa ser escrito e testado. Para uma equipe muito pequena, a divisão pode custar mais do que devolve; um armazenamento criptografado, permissões de armazenamento rígidas e uma recuperação de chave documentada podem ser um ponto de partida razoável.

O terceiro custo é um ciclo de depuração mais lento. Quando os engenheiros não conseguem restaurar a produção no staging, alguns bugs demoram mais para ser reproduzidos, porque o formato que os dispara ainda não está nos dados de teste. O hábito das fixtures compensa isso ao longo do tempo e é uma despesa real no começo. As cópias mascaradas acrescentam um pipeline de que alguém precisa ser responsável, e ele quebrará quando uma coluna for adicionada, e é por isso que a migração que adiciona uma coluna é onde sua classe é decidida (Capítulo 11).

O quarto custo é a retenção. Uma retenção mais longa mantém mais histórico recuperável e mantém mais dados sensíveis em mais lugares; uma retenção mais curta inverte as duas coisas. Não existe um valor certo para todos, e uma equipe que não escolheu um escolheu “para sempre”.

Por fim, o registro e o seu sweeper são sobrecarga. Um sweeper que exclui uma cópia de que alguém ainda precisava é um incidente por si só, então a exclusão deve vir depois de uma notificação e de um período de carência, e um responsável deve poder estender uma data de término uma vez, com um motivo. Um registro que nunca é aberto não protege nada, e o vínculo com a revisão de acessos é o que o mantém honesto.

Quando não construir tudo isso: uma empresa pequena, com um único banco de dados, um punhado de engenheiros e nenhum dado Confidential que mereça o nome, não precisa de aprovação de exports por faixas nem de uma chave dividida. Ela ainda precisa do inventário, de armazenamento restrito para os backups e de uma regra de que nada de produção chega

ao staging sem máscara. Faça essas três coisas, registre por escrito o que faria você fazer mais e revise essa anotação quando a equipe ou os dados mudarem.

Erros comuns

- **Estreitar o acesso ao primary e ignorar seus backups.** Há papéis, um gateway e um log de auditoria, e o bucket de backups é legível por todo o grupo de infraestrutura. É tentador porque backups são “operações”. O remédio é tratar a restauração como uma leitura de tudo, com aprovador próprio e log próprio.
- **Acreditar que “criptografado” significa “seguro”.** A criptografia em repouso protege o arquivo de quem está fora do caminho de restauração, e a chave fica ao lado do bucket, nas mãos do mesmo administrador. Mantenha a chave separada e diga contra o que ela não protege.
- **Excluir dumps antigos antes de existirem backups de verdade.** O dump era a única cópia de uma tabela de alguém, e removê-lo removeu o único caminho de volta. Construa e teste os backups primeiro e depois aposente os dumps.
- **Restaurar em uma rede onde os logins antigos funcionam.** A cópia restaurada traz os papéis e as credenciais do seu dia, inclusive os aposentados desde então. Restaure em isolamento, reaplique a definição de acesso e substitua o login `ledgerly_access_admin` que o backup trouxe por um novo, em vez de manter qualquer uma das credenciais antigas.
- **Tratar uma cópia mascarada como pública.** Os valores estão mascarados e a cópia é compartilhada amplamente, e então um texto livre ou uma combinação rara de campos aponta para um único cliente. Trate-a como Confidential, com um público menor e uma data de término, até que seja revisada.
- **Atualizar o staging a partir da produção como exceção pontual.** A etapa de mascaramento falhou, o release estava esperando e o dump estava ali. Faça a atualização falhar fechado, para que os dados antigos permaneçam e nada recorra a uma fonte bruta.
- **Deixar uma ferramenta de BI se conectar com um papel de produção.** Um login por trás de, digamos, cem espectadores, com um cache

que ninguém listou. Conecte-a à cópia de analytics com papel próprio e registre os seus extratos.

- **Aprovar todo export em segundos.** A aprovação existe e não reduz nada. Reserve o aprovador para a faixa que merece uma decisão e observe se alguém alguma vez recusa.
- **Dar a uma cópia um responsável e nenhum fim.** O registro está completo e nada sai dele. Toda linha precisa de uma data em `ends` e de um `sweeper` que aja sobre ela, com uma notificação antes.

Pontos-chave

- Uma porta lateral leva os dados sem os controles do sistema de onde vieram, porque os controles de acesso vivem no banco de dados e não no arquivo.
- O poder de restaurar um backup é o poder de ler tudo o que há nele, então o acesso a backups merece a mesma revisão que o papel humano mais privilegiado.
- Criar, restaurar e excluir backups devem ser poderes de principais diferentes, e a chave de criptografia deve ser mantida separada dos administradores do armazenamento, com um caminho de recuperação testado.
- Uma cópia restaurada volta com os papéis e as credenciais do dia em que foi feita, então ela começa isolada, tem a definição de acesso replicada e recebe um login de administrador próprio.
- Um export para o fornecedor é controlado pelo grant de `app_vendor_export`, não pela consulta, de modo que o arquivo não pode conter uma coluna que o papel não consegue ler, e um manifesto permite que o arquivo seja verificado.
- Os dados de teste devem ser mascarados ou sintéticos, uma cópia mascarada é tratada como Confidential até ser revisada, e o staging deve negar mensagens de saída por padrão e falhar fechado quando o mascaramento falhar.
- Toda cópia precisa de um responsável e de uma data de término em um registro que seja revisado junto com os acessos, porque uma porta que ninguém possui é uma porta que ninguém fecha.

Cópias, exports e restaurações agora têm responsável e data de término, mas um projeto só vale tanto quanto a evidência de que é seguido. O próximo capítulo trata da auditoria: onde o log de auditoria vive, o que entra nele e como as revisões de acessos que este livro menciona o tempo todo realmente acontecem.

PARTE V

Comprovar e operar

Capítulo 14 — Auditoria e responsabilização

Imagine a Ledgerly em uma terça-feira de manhã. Um cliente perguntou quem, na empresa, consultou a conta de repasses dele no mês passado, e o contato de segurança encaminha a pergunta para Alex. Parece uma simples consulta. Alex começa pela lista de sessões do access gateway, que mostra que Robin manteve uma sessão break-glass no primary por quarenta minutos. Depois abre o log do banco de dados, que nomeia um login de sessão, mas não o motivo da sessão. Em seguida pesquisa `audit_events`, que registra as ações de negócio executadas pela aplicação e não diz nada sobre o que uma pessoa digitou.

As três fontes concordam na maior parte da história e discordam no resto. O relógio do gateway está dois minutos adiantado em relação ao do banco de dados. Um login de sessão aparece no log do banco sem nenhuma entrada correspondente no gateway, e ninguém sabe dizer se é um login de serviço que a lista esqueceu ou uma conexão que chegou por um caminho que ninguém projetou. Alex consegue dizer o que provavelmente aconteceu, mas não consegue demonstrá-lo, nem demonstrar que ninguém com direitos de administrador de banco de dados editou a evidência.

Este capítulo atende ao pilar da responsabilização da tese e acrescenta o último passo do estado-alvo antes dos incidentes: o acesso é registrado em um armazenamento que o sistema acessado não consegue reescrever, e revisado em intervalos regulares. No modelo de cinco partes (**Identidade, Grant, Caminho, Execução, Registro**), este é o capítulo sobre o Registro. A Identidade diz quem agiu; o Registro é o que permanece, e se ele liga a instrução à pessoa, à requisição e ao resultado. A pergunta aqui é esta: o que uma instrução deve deixar para trás, onde isso deve ficar e como descobrir que a evidência está incompleta antes que alguém pergunte?

O problema e por que ele é difícil

No estado inicial, a pergunta não tem resposta. O log do banco de dados nomeia um papel compartilhado e nenhuma pessoa, e fica em hosts que os administradores controlam. Os capítulos anteriores resolveram a metade da identidade: um login por sessão, uma requisição por grant, um runner para as escritas aprovadas. Cada um deles acrescentava ao registro por meio da mesma chamada, `auditLogAppend`, sem dizer para onde as entradas vão. Este capítulo responde a isso.

A dificuldade é que a Ledgerly já tem três registros, e cada um sabe algo diferente. Nenhum deles é o log de auditoria como o livro o define: o registro de quem fez o quê com quais dados, mantido onde o autor da ação não pode alterá-lo.

Registro	O que ele sabe	O que ele não consegue dizer	Quem pode alterá-lo
Registro de sessão do gateway	A pessoa, a requisição, o motivo, a instrução como foi submetida	O que o banco de dados realmente fez; se existia outro caminho	A equipe de plataforma, por meio do armazenamento do próprio gateway
Log do banco de dados	O login, o horário, o texto da instrução se foi registrado, erros	A pessoa por trás do login, a requisição, o motivo	Quem administra o host do banco de dados
<code>audit_events</code>	A ação de negócio, o principal que agiu, o sujeito	As instruções que a produziram; qualquer coisa que a aplicação não escreveu	Ninguém por <code>UPDATE</code> , mas <code>ledgerly_owner</code> pode alterar a tabela em uma migração

A resposta óbvia é aumentar o nível de logging: registrar toda instrução no banco de dados e guardá-la por muito tempo. Isso acrescenta um quarto registro e não resolve nenhuma das três lacunas. Um log que o sistema acessado escreve sobre si mesmo só é tão confiável quanto seus administradores, e um log que ninguém lê é teatro. Um log de instruções é, ele próprio, uma cópia de dados sensíveis, porque as pessoas digitam literais nas condições; o Capítulo 12 alertou sobre isso, e este capítulo é responsável pelo mecanismo.

O registro existe para encontrar a lacuna no sistema, não para classificar as pessoas que passaram por ele. Quando um alerta dispara na sessão de Robin, a primeira pergunta é o que o design tornou necessário, não o que Robin fez de errado.

Uma instrução, um registro

Um registro só é útil se puder responder a uma lista de perguntas escrita com antecedência. Escreveria a lista primeiro e derivaria os campos dela, porque um armazenamento de auditoria projetado a partir das linhas de log que o gateway por acaso produz responde às perguntas que essas linhas por acaso permitem.

O gateway escreve duas entradas por instrução, porque o Capítulo 9 exige que a intenção seja escrita antes da execução da instrução e o resultado depois dela. Juntas, formam o **registro da instrução**; as duas entradas compartilham um `statement_id`, e a segunda preenche os campos que eram desconhecidos no início. Um único formato serve para as duas e para os registros que outros principais escrevem.


```

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

```

O tipo `event` carrega tudo o que não é uma instrução: a abertura de uma sessão, um grant, uma revogação, uma rotação, uma restauração, o início de uma migração. Essas são as chamadas `auditLogAppend` e `auditOrQueue` dos capítulos 3, 4, 7, 8, 10, 11, 12, 13 e 15, e a coluna `action` distingue um grant de uma revogação ou de uma contenção. Elas usam a mesma cadeia e os mesmos campos de identidade, de modo que uma única consulta pode seguir uma pessoa desde uma requisição, passando por um grant e pelas instruções que o usaram, até a revogação.

A tabela só vale a pena ser construída se responder às perguntas que as pessoas fazem. A segunda lista é a que colaria ao lado do esquema.

Pergunta	Campos que respondem
Quem executou esta instrução?	<code>principal</code> , <code>on_behalf_of</code> , <code>login</code>
Com quais grants e qual aprovação?	<code>roles</code> , <code>request_id</code> , <code>approver</code> , <code>reason</code>
O que ela pedia?	<code>statement_class</code> , <code>fingerprint</code> , <code>literals_hash</code> , <code>tables</code> , <code>columns</code>
O que ela fez?	<code>outcome</code> , <code>rows_affected</code> , <code>rows_returned</code>
Ela tocou em dados Restricted?	<code>columns</code> classificadas por classe de dados, com <code>rows_returned</code>
Quando, e por quanto tempo?	<code>started_at</code> , <code>ended_at</code>
Falta alguma coisa ao redor dela?	<code>stream</code> , <code>sequence</code> , <code>previous_hash</code> , a intenção sem resultado
Esta entrada está como foi escrita?	<code>entry_hash</code> , verificado por <code>verifyChain</code> abaixo

Dois campos carregam a maior parte do peso. `principal` e `on_behalf_of` impedem que o rastro termine em uma máquina, como o Capítulo 9 exige para a automação. O `request_id` é a chave de junção para tudo o que os capítulos 8 e 10 registraram.

Um registro sem `request_id` continua válido. Uma sessão `human_readonly` de rotina não tem requisição, e o registro diz isso deixando o campo vazio. Um `request_id` vazio em uma sessão que detém as triplas `human_jit_write` ou `human_breakglass` é um achado, e as verificações mais adiante neste capítulo o procuram.

Um armazenamento que o sistema acessado não consegue reescrever

O lugar onde as entradas ficam importa mais do que o formato delas. O Capítulo 1 definiu a observabilidade como um registro que fica em um lugar que o principal não pode alterar. Os principais que importam aqui são os mais poderosos: os administradores do banco de dados, a equipe de plataforma que opera o gateway e `ledgerly_access_admin`, cujos três lo-

gins podem conceder quase qualquer coisa dentro dos papéis que ele administra. Se qualquer um deles puder editar o registro, o registro é evidência apenas do que escolheram deixar.

Construiria o armazenamento com cinco propriedades, nesta ordem.

1. **Uma conta separada.** O armazenamento de auditoria vive em uma conta de nuvem, projeto ou grupo de hosts em que os administradores do banco de dados, os operadores do gateway e os três logins de `ledgerly_access_admin` não têm nenhum direito. Alguém precisa administrá-lo; é um grupo pequeno, sem direitos de administrador de banco de dados, e o uso que ele faz do armazenamento é visível para alguém de fora.
2. **Um produtor somente de escrita.** Cada produtor (o gateway, o runner, o credential broker, o pipeline, o job de revogação, as ferramentas de restauração) detém a própria credencial, que pode chamar uma única operação: `append`. Ela não pode ler, atualizar nem excluir. Uma credencial de produtor vazada pode acrescentar entradas falsas, o que uma cadeia torna detectável, e não pode remover as verdadeiras.
3. **Um serviço de `append` que mantém a cadeia.** Os produtores não calculam hashes. Um pequeno serviço dentro da conta de auditoria carrega o head do stream, encadeia a nova entrada e armazena ambos. Um produtor que calculasse o próprio elo também poderia recalculá-lo tudo o que vem depois dele.
4. **Uma retenção que o administrador da conta não consegue encurtar.** Quando o armazenamento oferece um bloqueio de retenção de escrita única, use-o durante o período de retenção, para que excluir uma entrada antes do fim desse período não seja possível para ninguém na conta. Quando não oferece, o enunciado honesto é que o administrador pode excluir, e a cadeia e a âncora abaixo são o que tornam uma exclusão visível.
5. **Uma âncora.** Uma vez por dia, o serviço de `append` envia o hash do head de cada stream a um segundo lugar, de propriedade de outro grupo: o sistema dos contatos de segurança ou um segundo armazenamento. Sem uma âncora, uma pessoa com controle total do armazenamento poderia reescrever um stream a partir de certo ponto, recalculando todos os hashes seguintes. Com uma âncora, o head reescrito difere dela.

```

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 }

```

Duas sessões podem fazer append no mesmo stream ao mesmo tempo, por isso o head é lido sob um bloqueio dentro da transação; sem ele, ambas leriam o mesmo head, escreveriam o mesmo número de sequência e bifurcariam a cadeia, e o verificador então relataria uma adulteração que nunca aconteceu. A regra é um único escritor por stream de cada vez, imposta por esse bloqueio.

O verificador é um job separado, com uma credencial somente de leitura na mesma conta. O trabalho dele é a parte que transforma um armazenamento em evidência:

```

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)

```

Agora o limite honesto, que colocaria no documento de design como uma frase própria. Este design dá evidência, não prevenção. Ele não impede que uma instrução seja executada, e não impede que alguém com poder suficiente sobre a conta de auditoria destrua o armazenamento. Ele torna a alteração detectável e faz com que as pessoas capazes de alterá-la formem uma lista curta e separada. A detecção também depende de alguém ler a saída de `verifyChain`, de modo que o verificador é um job agendado com um heartbeat próprio, tratado na próxima seção.

O custo é real. A conta de auditoria é mais um sistema a manter disponível, e os capítulos 8 e 9 já aceitaram sua pior consequência: um grant, uma declaração de break-glass ou uma instrução é recusado quando sua intenção não pode ser escrita. Uma indisponibilidade do armazenamento de auditoria, portanto, bloqueia o trabalho privilegiado, e o bloqueia justamente quando alguém pode estar esperando por uma emergência. Esse é o preço de um registro que não depende de boa vontade.

Aviso: Não coloque o armazenamento de auditoria no banco de dados que ele descreve, nem mesmo em um esquema com proprietário próprio. Quem pode administrar esse banco de dados pode alterar as entradas, e um papel somente de `INSERT` não muda isso. A única exceção estrutural que o livro já lista, a de que `ledgerly_owner` pode alterar `ledger_entries` e `audit_events` em uma migração, é o motivo de `audit_events` ser o registro de negócio da aplicação e não o log de auditoria.

Encontrando as lacunas

Uma cadeia prova que as entradas presentes não foram alteradas. Ela não prova que toda instrução produziu uma entrada. A segunda família de verificações procura o que deveria existir e não existe. Há quatro tipos de lacuna, e cada um tem um significado diferente, por isso a verificação os relata separadamente.

Buracos de sequência. Um stream é sem lacunas por construção, então um buraco significa que o serviço de append perdeu uma entrada ou que alguém removeu uma. É o achado mais raro e o mais grave.

Intenções sem resultados. O gateway escreve uma intenção antes de uma instrução e um resultado depois. Uma intenção sem resultado após a mais longa das instruções que o timeout permite significa que o gateway falhou entre as duas, que o resultado está esperando na fila de retry ou que a instrução ainda está em execução. Só a terceira hipótese é inofensiva, e ela não deveria durar além do timeout.

Sessões em apenas um dos lados. Todo login do banco de dados deveria ter uma origem conhecida. Para humanos, a origem é uma sessão do gateway. Para serviços, é um lease do credential broker do Capítulo 4. Para migrações, é uma execução do pipeline. Um login no log do banco de dados que não tenha nenhuma das três é um caminho que ninguém projetou, e uma sessão do gateway que alega instruções que o banco de dados nunca viu é um registro que descreve algo que não aconteceu.

Produtores silenciosos. Um stream que para de escrever pode significar uma semana tranquila, um produtor morto ou um produtor que alguém desligou. A verificação compara cada stream com uma cadência 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)
```

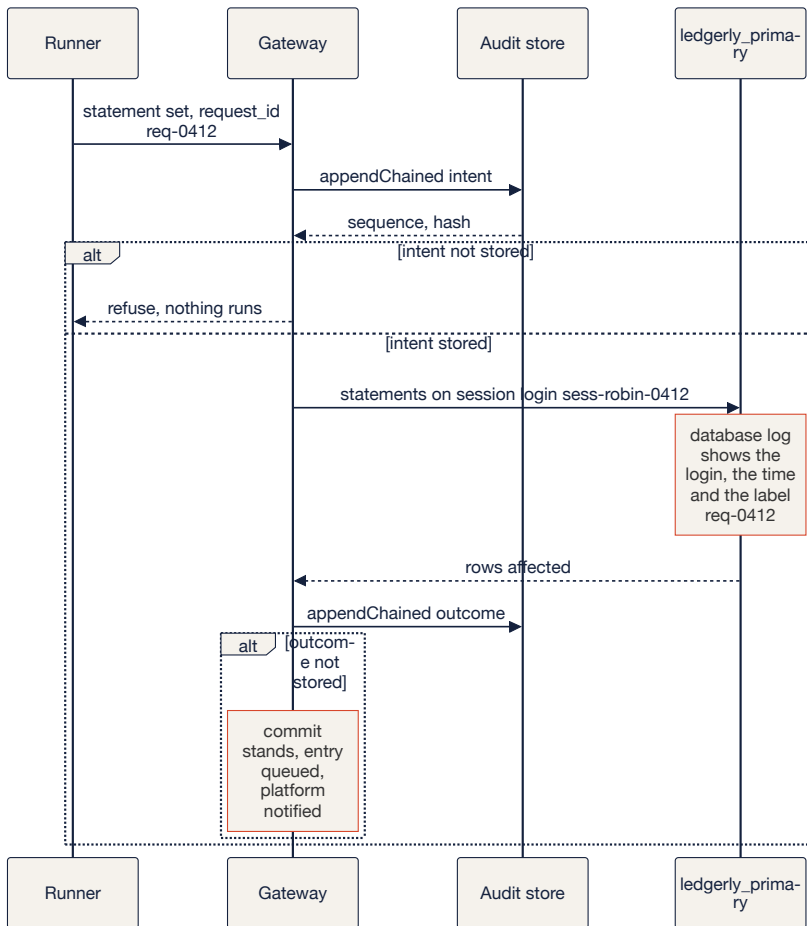
O log do banco de dados chega à conta de auditoria por meio de um agente de envio com a própria credencial somente de escrita. Alguém com acesso ao host pode parar esse agente, e é para isso que serve a verificação de stream silencioso: um agente parado deixa um stream sem entradas, e isso gera um achado próprio.

Juntando três registros

O registro do gateway, o log do banco de dados e `audit_events` são produzidos por três sistemas que nunca compartilham uma transação. Juntá-los é o que transforma os casos da manhã de Alex de palpites em uma linha do tempo, e as chaves de junção precisam ser escolhidas antes do incidente, porque não podem ser acrescentadas depois.

Três chaves fazem o trabalho. O **login** aparece no log do banco de dados e, como um login de sessão corresponde a exatamente uma pessoa ou uma instância de workload, ele liga uma linha de log a um registro de sessão. O **request id** é escrito pelo gateway no registro de auditoria e pelo runner no campo `request_id` do próprio armazenamento de auditoria; ele liga a instrução à aprovação que a permitiu. O runner nunca escreve em `audit_events`, o que a matriz do Capítulo 9 lhe nega. As linhas de `audit_events` que carregam o correlation id introduzido no Capítulo 3 vêm do caminho de produto, em que `admin-console` chama `payments-service`, e se juntam à mesma requisição por meio desse id. O **rótulo de conexão** do Capítulo 5 pode levar o request id para o log do banco de dados nas conexões mediadas. Esse rótulo é definido por quem abre a conexão. Na execução mediada, o gateway a abre com o request id da requisição aprovada, então ele é confiável; em uma session handout, o cliente da pessoa poderia defini-lo como qualquer coisa, então o login continua sendo a chave e o rótulo continua sendo uma pista.

O diagrama mostra uma correção de dados aprovada e onde cada registro é escrito; o runner chega ao primary apenas por meio do gateway. Se a escrita do resultado falhar, o commit permanece e a lacuna é detectável.



O que o registro não deve conter e quem audita os auditores

Um registro de instrução contém o que as pessoas digitaram. Uma pessoa que procura um cliente pelo número da conta colocou o número da conta na instrução, e um log que armazena a instrução copiou um valor Restricted para um lugar sem proteção em nível de coluna. O alerta do Capítulo 12 se aplica integralmente, então quem constrói o registro precisa decidir o que é mantido.

Armazenaria a instrução em duas partes. O **fingerprint** é a instrução com cada literal substituído por um marcador, de modo que `UPDATE invoices SET status = ? WHERE id IN (?)` registra o formato e nada mais. O **literals hash** é um hash com chave dos valores literais, que permite a um

investigador testar um valor suspeito contra o registro sem que o registro o contenha. A chave importa: literais como um status ou um sufixo curto de conta têm baixa entropia, e um hash sem chave poderia ser revertido testando candidatos. A chave fica com a conta de auditoria, separada dos produtores, e a ferramenta do investigador pede uma comparação e nunca a chave.

```
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
```

As últimas linhas são as que mais importam. Um fallback que armazenasse a instrução bruta quando o parser falha manteria justamente o caso em que um literal tem mais chance de aparecer, porque as instruções incomuns são aquelas em que um parser falha. Não armazenaria nada e deixaria a recusa falar. O próprio armazenamento é classificado no mínimo como Confidential, porque descreve quem olha o quê, e tem a própria lista de leitores e a própria retenção. O quanto uma empresa pode registrar sobre os próprios funcionários é uma pergunta para o seu pessoal jurídico e de conformidade.

Quem audita `ledgerly_access_admin`

`ledgerly_access_admin` é o papel que pode criar logins e conceder qualquer coisa dentro dos papéis que administra, usado por meio de três logins: a etapa do pipeline que aplica a definição de acesso (Capítulo 3), o credential broker (Capítulo 4) e o gateway (Capítulo 7). O Capítulo 3 dis-

se que toda ação vai para o log de auditoria; o mecanismo é que cada um dos três é um produtor com o próprio stream, de modo que o registro nomeia qual login agiu. Cada `CREATE LOGIN`, `GRANT` e `REVOKE` que ele emite é uma entrada `event`, escrita antes da mudança, com a requisição ou a execução que a causou.

Isso dá uma verificação que fecha o ciclo. O catálogo do banco de dados é a verdade sobre quais grants existem, e o armazenamento de auditoria é a verdade sobre o que foi concedido. Um grant que aparece no catálogo sem uma entrada correspondente foi feito fora dos três produtores, e esse é o grant das 2 da manhã que a verificação de drift do Capítulo 3 foi criada para encontrar. As duas verificações são a mesma coisa vista de pontas opostas: o drift compara o catálogo com o código, e esta compara o catálogo com o registro. Nenhuma das duas repara nada; ambas relatam.

Migrações

O pipeline de migração escreve no armazenamento de auditoria antes de executar uma migração, como o Capítulo 11 exige, e de novo quando ela termina ou falha. A razão da entrada externa é que a ferramenta de migração também registra o que aplicou em uma tabela dentro de `ledgerly_primary`, e `ledgerly_migrator` assume `ledgerly_owner`, que pode alterar essa tabela. Um registro que o autor da ação pode reescrever é uma conveniência e não evidência. A regra é que uma migração conta como aplicada quando a entrada externa de início e o hash do commit existem no armazenamento de auditoria, e uma linha na tabela de migrações sem entrada externa correspondente é relatada pela mesma comparação entre catálogo e registro, como uma mudança que contornou o pipeline.

Alertas que alguém vai atender

Um registro responde a perguntas depois do fato. Algumas dessas perguntas valem ser feitas continuamente, e um alerta é a forma de fazer isso. O teste que aplicaria a toda regra é se alguém foi designado para atendê-la e sabe o que fazer.

A Ledgerly começa com três regras, cada uma explorando algo que o registro sabe e as outras fontes não.

```

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)

```

Os limites são ilustrações; o `nightRowThreshold` certo é o que colocar meia dúzia de eventos por semana diante de uma pessoa. A segunda regra é a verificação independente da declaração do Capítulo 8: aquele capítulo envia um alerta quando uma pessoa declara a emergência, e esta regra encontra uma sessão que detém o papel sem declaração, o que é um grant que a contornou. A terceira regra quase nunca deveria disparar.

`human_readonly` não pode ler colunas Restricted e uma requisição JIT não pode nomear nenhuma, portanto uma leitura humana de dados Restricted é ou uma sessão break-glass declarada ou um grant que não corresponde à matriz, e o segundo caso é um achado de design.

Cada regra precisa de um responsável nomeado e de uma primeira ação: o contato de segurança pergunta qual era a tarefa, o engenheiro de plataforma de on-call confirma a declaração e abre o item de revisão, a equipe de plataforma revoga um grant que contornou o processo. Uma regra reconhecida sem que ninguém olhe é teatro de segurança.

A revisão de acessos

Os alertas observam eventos. A **revisão de acessos** é a verificação agendada de que o acesso de cada principal ainda é necessário, e é o controle que pega a falha lenta que nenhum evento mostra: um acesso que estava certo quando foi concedido e silenciosamente deixou de estar. As pessoas mudam de equipe, um serviço para de ler uma tabela, um papel mantém um privilégio que seu workload não usa há meses. O Capítulo 3 disse que um grant errado, corretamente declarado, passa pela verificação de drift, e que a revisão de acessos é o lugar onde esse julgamento pertence.

A revisão compara o que foi concedido com o que foi usado. O registro fornece a segunda lista, e é isso que faz dela uma decisão e não um formulário para assinar.

```
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)
```

Três regras governam como usaria a saída dela. Primeiro, a função propõe e nunca revoga, pela razão que o Capítulo 3 deu: uma revogação automática de um grant de que alguém depende é uma nova maneira de quebrar a produção. Segundo, o `period` precisa ser mais longo que o ciclo mais longo do workload. Um grant usado por um relatório trimestral fica sem uso na maioria dos meses, então uma revisão mensal proporia revogá-lo em dois meses de cada três. Terceiro, o uso é associado por privilégio e não apenas por tabela. Um papel que faz select em uma tabela toda noite

e nunca a atualizou é candidato a ser reduzido de `S, U` para `S`, e o registro tem o que é preciso, porque cada registro carrega a classe da instrução e as tabelas que ela tocou.

A revisão tem uma segunda entrada que o registro não consegue fornecer: coisas que não são grants. O registro de cópias do Capítulo 13 pertence à pauta, assim como a lista de quem pode aprovar requisições JIT e a lista de quem pode ler o armazenamento de auditoria. A pergunta para cada um é a mesma: a pessoa ou o sistema ainda precisa disso, e alguém ainda é responsável por isso? O Apêndice tem um modelo.

Evidência, não conformidade

Os regimes de proteção de dados e de cartões de pagamento tendem a fazer uma família de perguntas sobre o acesso: quem pode alcançar os dados, quem fez o quê, quanto dura o acesso, onde estão as cópias e o que acontece em uma emergência. Um auditor que trabalha sob um desses regimes geralmente vai querer ver que um controle existe, que ele foi executado e que seus resultados foram examinados. O seu pessoal jurídico e de conformidade é responsável pela interpretação de qualquer regime. Esta seção não afirma mais do que a engenharia acima pode entregar, que é evidência: registros que respondem a perguntas e uma forma de mostrar que não foram alterados.

Nada neste capítulo deixa uma empresa em conformidade com coisa alguma. A tabela lista as perguntas que um auditor costuma fazer, o controle que responde a cada uma, o artefato que ele deixa e o capítulo que o constrói; o Apêndice a repete como uma correspondência.

Pergunta	Controle	Artefato	Ca- pí- tulo
Quem pode alcançar os dados de produção, e por qual caminho?	Gateway como única porta, esque- ma de papéis como código, verifi- cação de drift	Lista de regras de rede, definições de papéis, relatório de drift	3, 7
Quem executou uma determinada instrução, com quais grants e qual aprovação?	<code>access_audit_record</code> com re- quest id e aprovador	O registro e o relatório de <code>verifyChain</code>	14

Pergunta	Controle	Artefato	Capítulo
Quanto dura o acesso elevado, e quem o aprovou?	Requisição JIT com aprovador e <code>expires_at</code> , job de revogação	Registros de requisições, log de revogações, verificação de grants vencidos	8
Como os dados sensíveis são identificados e limitados?	Classificação no esquema, mascaramento, colunas retidas	O catálogo, a verificação da carga mascarada	12
Onde estão as cópias dos dados de produção, e quem é responsável por cada uma?	Registro com responsável e data final, <code>sweeper</code>	Registro, log do <code>sweeper</code>	6, 13
Quem altera o esquema, e quem o revisou?	Pipeline de migração, branch protegida, entrada externa de início	Registros de migração associados a commits e entradas de auditoria	11
O acesso ainda é necessário?	Revisão de acessos com revisores nomeados	Revisão concluída com decisões	14
O que acontece em uma emergência?	Break-glass com alerta e revisão obrigatória	Registros de revisão, histórico de alertas	8, 15
Quem administra o próprio acesso?	Logins de <code>ledgerly_access_admin</code> como produtores separados	Seus streams e a comparação entre catálogo e registro	3, 14

Trade-offs e custos

O registro tem uma conta, e a maior parte dela é paga em dois lugares.

Armazenamento e atenção. Um registro por instrução, duas vezes, cresce com o tráfego, e o armazenamento precisa de retenção, backups e lista de leitores próprios. O custo maior é a atenção. Um armazenamento que ninguém consulta é teatro de segurança caro, de modo que a amostra semanal de sessões elevadas do Capítulo 7, os alertas e a revisão fazem parte do preço e não são extras.

Uma nova dependência no caminho crítico. Como a intenção é escrita antes da execução da instrução, uma indisponibilidade da auditoria bloqueia o trabalho privilegiado. A alternativa, executar as instruções e registrar depois, torna o registro opcional exatamente quando ele importa.

Manteria a recusa para grants, break-glass e escritas, e consideraria uma escrita em fila para as leituras `human_readonly` de rotina, com um teto para o quanto a fila pode crescer, se a equipe decidir que uma indisponibilidade de leitura custa mais que uma lacuna. Essa é uma decisão para a equipe de plataforma tomar uma vez e registrar por escrito.

Quando não construir tudo isso: uma empresa com um punhado de engenheiros, um banco de dados e nenhum gateway ainda não deveria começar por um serviço de append, uma âncora e uma conta separada. Comece por três coisas: registrar a sessão de cada pessoa sob o login dela, enviar o log do banco de dados para fora do host, para um lugar de onde os administradores não possam excluí-lo, e colocar uma pessoa nomeada em uma revisão mensal. Acrescente a cadeia e a âncora quando o número de pessoas que poderiam alterar o log tornar a “confiança” uma resposta inadequada.

Erros comuns

- **Tratar o log do banco de dados como o log de auditoria.** Ele é a fonte mais completa e fica sob o controle dos administradores. Envie-o para outro lugar e use-o como uma das entradas da junção, não como o registro.
- **Uma cadeia de hashes sem um armazenamento separado.** A cadeia parece uma prova e pode ser recalculada por quem consegue escrever no armazenamento. Coloque o armazenamento em outra conta, mantenha a cadeia no serviço de append e ancore o head.
- **Armazenar a instrução bruta.** É tentador porque facilita a investigação, e copia valores Restricted para um armazenamento sem proteção por coluna. Armazene o fingerprint e um hash com chave, e recuse o fallback para o texto bruto.
- **Alertas sem responsável.** Um canal que ninguém foi designado para atender vira ruído. Dê a toda regra um responsável e uma primeira ação.
- **Auditar tudo, menos os auditores.** O papel que concede acesso é aquele cujas ações mais importam. Dê a ele os próprios produtores e compare o trabalho dele com o catálogo.

Pontos-chave

- O log de auditoria é o registro de quem fez o quê com quais dados, mantido onde o autor da ação não pode alterá-lo, e o registro do gateway, o log do banco de dados e `audit_events` são três entradas dele, não ele próprio.
- Um registro de instrução é construído a partir de uma lista de perguntas escrita primeiro, e seus campos de identidade terminam em uma pessoa mesmo quando uma máquina agiu, por meio de `principal` e `on_behalf_of`.
- A evidência de adulteração vem de uma conta separada, produtores somente de escrita, um serviço de append que mantém a cadeia e uma âncora de propriedade de outra parte, e ela dá evidência e nunca prevenção.
- Uma cadeia mostra que as entradas presentes não foram alteradas, por isso verificações separadas procuram buracos de sequência, intenções sem resultados, logins do banco de dados sem origem conhecida e produtores que ficaram em silêncio; os três registros se juntam pelo login e pelo request id, e o rótulo de conexão é apenas uma pista.
- O registro nunca deve se tornar uma cópia dos dados que descreve, por isso as instruções são armazenadas como fingerprints com um hash com chave dos literais, e o armazenamento é classificado e restrito como os dados.
- Os alertas precisam de um responsável e de uma primeira ação, e a revisão de acessos compara grants com uso em um período mais longo que o ciclo do workload, de modo que as revogações sejam propostas a partir de evidência.
- Evidência não é conformidade: a correspondência entre pergunta, controle, artefato e capítulo mostra o que o sistema consegue provar, e o seu pessoal jurídico e de conformidade é responsável pelo que essa prova significa sob qualquer regime.

O registro agora existe, está unido e é vigiado. O que resta é o dia em que ele é necessário sob pressão, quando quem pergunta já não é um auditor, mas um engenheiro de on-call com uma credencial vazada. O próximo capítulo trata do acesso durante incidentes e usa o registro construído aqui para descobrir o que um principal comprometido de fato tocou.

Capítulo 15 — O acesso durante incidentes

Imagine um sábado à tarde na Ledgerly. Sam, do suporte, escreve no canal de on-call: um laptop da empresa foi levado de um carro estacionado, e uma sessão do gateway pode ter ficado aberta nele. Robin está de on-call e lê a mensagem duas vezes. Não se sabe de nada errado, mas “pode ter ficado aberta” é o tipo de frase que precisa ser respondida, não ponderada.

Robin precisa de três coisas, e a ordem importa. A primeira é parar tudo o que essa sessão, e qualquer login por trás dela, ainda pode fazer. A segunda é descobrir o que ela fez enquanto pôde. A terceira é manter o negócio funcionando enquanto as duas primeiras acontecem, o que significa que outras pessoas ainda precisam do banco de dados, algumas com urgência, e os aprovadores de costume estão em casa.

Este capítulo trata o acesso em incidentes como algo a ser projetado e ensaiado. Ele exercita os três pilares ao mesmo tempo: a autorização, quando grants são removidos ou acrescentados às pressas; a execução, quando os reparos precisam rodar sob pressão; e a responsabilização, quando o registro precisa dizer o que aconteceu. No modelo de cinco partes, a Identidade encontra os logins, o Grant e o Caminho os cortam, a Execução carrega as escritas de emergência e o Registro mede o dano. Este capítulo acrescenta o último passo do estado-alvo da Ledgerly: o acesso em incidentes tem um caminho ensaiado, e a revogação é uma rotina. A pergunta é como a Ledgerly interrompe o acesso, mede o que foi alcançado e continua funcionando, sem reabrir as portas que os capítulos anteriores fecharam.

O problema e por que ele é difícil

Um incidente é o momento em que todo controle anterior parece um obstáculo. A aprovação que era razoável em uma terça-feira tranquila é uma espera em um sábado, o gateway que registra tudo é mais uma coisa que

pode estar fora do ar, e o atalho mais limpo da sala é um login que pula tudo isso. A resposta habitual é construir esse login com antecedência “para emergências”. É a senha de administrador compartilhada de novo, com um nome melhor.

Três coisas mudam sob pressão. O tempo é curto, então os passos que dependem de uma pessoa lembrar são pulados. O conhecimento é parcial, então a primeira estimativa de escopo está errada, para mais ou para menos. E os atores são incomuns: pessoas que raramente tocam o banco de dados de repente precisam fazê-lo, e as que o tocam todo dia estão cansadas.

A pergunta de design, portanto, é quais controles relaxam e quais não. Manteria uma regra: a urgência muda a velocidade, nunca a identidade, nunca a instrução que é executada e nunca o registro. Os períodos de espera e a ordem dos aprovadores podem encolher. Um principal agindo sob o próprio login, uma aprovação vinculada a um hash de instrução e uma entrada no log de auditoria não podem. Se um controle não pode ser encurtado sem remover uma dessas três coisas, o controle está errado, não a regra.

A velocidade precisa ser construída. Uma revogação que exige uma dúzia de passos manuais será feita na ordem errada ou pela metade. A ordem é código, a consulta de escopo é código, e a reconciliação que diz se o incidente deixou algum acesso para trás é código. Um runbook que aponta para isso é um procedimento; um que o substitui por julgamento é uma esperança.

Contendo um comprometimento

Contenção aqui significa interromper o acesso de um principal, seja a causa um laptop perdido, uma credencial vazada ou um workload que se comporta de um modo que ninguém esperava. Ela não decide a causa. O Capítulo 4 separou duas perguntas que o pânico tende a misturar: como impedir que uma credencial funcione e o que ela fez enquanto funcionava. A contenção responde à primeira, e a ordem abaixo mantém a segunda respondível.

A ordem é: revogar grants, encerrar sessões, rotacionar credenciais, delimitar o dano a partir do log de auditoria e, por fim, registrar e notificar. Cada passo está onde está por um motivo.

Revogar grants vem primeiro porque uma sessão sem privilégios não pode fazer nada, e uma reconexão não obtém nada. Encerrar sessões vem em segundo lugar, por causa do trabalho em andamento e das conexões abertas que sobrevivem a um grant revogado (o Capítulo 4 fez a mesma observação sobre credenciais expiradas). A rotação vem em terceiro porque interrompe novos logins e nada faz pelos que já estão abertos. A delimitação é somente de leitura e precisa do registro, não do acesso, então espera até que o acesso tenha desaparecido. A notificação é a última, mas o registro de cada passo não: cada passo é acrescentado assim 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
        auditOrQueue({ 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
```

Dois detalhes merecem uma nota. A função devolve os passos não concluídos e não lança exceção, porque um principal contido pela metade é um estado a ser acompanhado, e uma exceção encerraria o único processo que sabe dele. E `auditOrQueue` é o mesmo helper do Capítulo 7: uma indispo-

nibilidade do log atrasa a entrada e nunca a remoção do acesso. A assimetria é a mesma que o Capítulo 8 escolheu para a revogação: um grant sem entrada de log é uma exposição, e uma revogação sem ela é uma lacuna que pode ser preenchida depois.

O que cada passo significa depende do que o principal é. A tabela lista os quatro casos que a Ledgerly tem.

Principal	Revogar grants	Encerrar sessões	Rotacionar credenciais
Pessoa, logins de sessão	Remover as triplas <code>JIT</code> , <code>human_breakglass</code> e <code>human_readonly</code> de todo login de sessão que a pessoa detém	<code>endSession</code> para cada login; desativar a pessoa no provedor de identidade para que nenhuma nova sessão seja aberta (Capítulo 7)	Redefinir as credenciais e os fatores da pessoa no provedor de identidade; os logins de sessão são descartados, então não há senha de banco de dados para rotacionar
Serviço, credencial dinâmica	Encerrar o lease; o login vai junto (Capítulo 4)	O broker encerra as sessões como parte do encerramento do lease	Nada manual; a próxima requisição recebe um valor novo
Serviço, credencial estática em um par A/B	Remover o papel <code>app_*</code> do login vazado imediatamente, sem esperar o login antigo esvaziar	Encerrar as sessões restantes dele	Publicar um valor novo no login ocioso e trocar o ponteiro; reiniciar os serviços que não conseguem reler o secret

Principal	Revogar grants	Encerrar sessões	Rotacionar credenciais
Login de <code>ledgerly_access_admin</code>	Remover seus privilégios, o que interrompe o provisionamento para o principal que o usava	Encerrar suas sessões	Emitir uma nova credencial para esse principal; a pergunta de escopo passa a ser quais grants e logins nenhuma requisição explícita (a verificação de drift do Capítulo 3)

A linha estática é a que corresponde ao Capítulo 4. Uma credencial estática vazada não é uma rotação planejada, e o login vazado é cortado antes de a substituição estar pronta, o que custa uma indisponibilidade nos serviços que o usam. O Capítulo 4 já aceitou esse custo, e manteria a aceitação: uma credencial vazada que continua funcionando é o problema maior. Os serviços que precisam de reinício deveriam estar em uma lista antes do dia em que isso importar. No estado inicial, em que todo serviço compartilha um único login de administrador, o passo um é a indisponibilidade da empresa inteira, o que é o argumento mais forte para construir primeiro a revogação de rotina.

A última linha é a pesada. O login que administra o acesso pode conceder qualquer coisa dentro dos papéis que administra, então contê-lo impede o gateway, o pipeline ou o broker de provisionar qualquer coisa até que exista uma nova credencial. Esse é o preço de cortar o login com o alcance mais amplo, e uma equipe deve decidir com antecedência quem pode pagá-lo.

Delimitando o raio de impacto a partir do registro

Quando o acesso desaparece, a pergunta passa a ser o que ele fez. A resposta é uma única consulta sobre `access_audit_record`, cujo design o Capítulo 14 descreve, executada com os campos que esse capítulo define. A consulta é o raio de impacto de um principal desde o momento T em

que seu acesso pode ter sido usado indevidamente pela primeira vez: cada tabela que ele tocou, quantas linhas voltaram, quantas foram alteradas e quais dessas tabelas guardam dados 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;
```

Escolher `:logins` e `:since` é onde fica a maior parte do julgamento. Para uma pessoa, o gateway cria um novo login para cada sessão, então o conjunto é todo login de sessão que o principal teve desde T, não um só. Para T, use o primeiro momento em que o acesso pode ter sido usado indevidamente, como a última hora em que se sabe que o laptop estava nas mãos do proprietário, e não a hora em que alguém percebeu. Uma janela estreita demais produz uma resposta limpa para a pergunta errada.

O resultado só é tão bom quanto o registro, então três verificações vêm antes de qualquer um lê-lo.

1. `verifyChain` passa sobre a janela, como o Capítulo 14 o define. Se falhar, o resultado é evidência de algo diferente do incidente, e o incidente ganha um segundo registro.
2. A janela não tem buracos de sequência nem sessão do gateway sem sessão no banco de dados, ou o inverso, que é a detecção de lacunas do Capítulo 14. Um buraco dentro da janela é um trecho em que a resposta é desconhecida.
3. Os caminhos laterais do principal são verificados separadamente: restaurações, exports e cópias que ele requisitou (Capítulo 13), encontra-

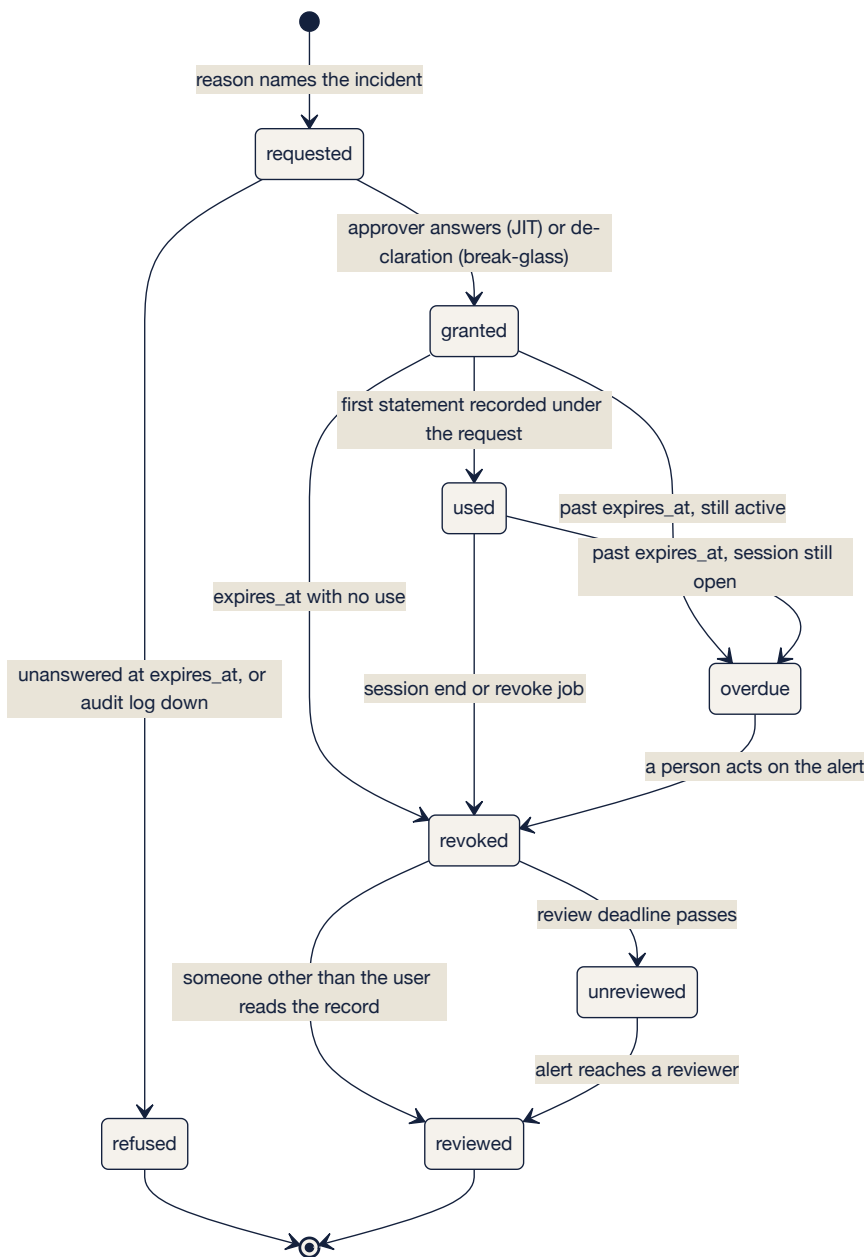
dos por meio do registro de cópias e não por esta consulta.

Se qualquer verificação falhar, o resultado é um limite inferior, e diria isso, com essas palavras, no registro do incidente. A consulta também tem limites próprios. `rows_returned` é uma contagem, não uma lista: ela diz que uma tabela Restricted devolveu certo número de linhas na janela e não consegue dizer quais clientes eram. Os campos `fingerprint` e `literals_hash` permitem perguntar se uma instrução conhecida foi repetida, mas os literais são removidos por design. Portanto o registro delimita a exposição e não a enumera. Um `rows_returned` diferente de zero em uma tabela Restricted é a linha de que os contatos de segurança e o pessoal jurídico e de conformidade do leitor precisam, e a interpretação do que isso significa, incluindo qualquer dever de notificação, é deles.

O ciclo de vida do acesso em incidentes

A contenção remove acesso. Um incidente também precisa de acesso acrescentado: alguém tem de olhar, e alguém pode ter de reparar. O risco nessa direção é o grant criado às pressas que ainda está lá na segunda-feira. A cura é um ciclo de vida com um fim visível, em que todo acesso criado para um incidente passa pelos mesmos cinco estados.

O diagrama é uma visão sobre os registros que o Capítulo 8 já mantém, não um novo campo de status. `granted` é `approved` no registro da requisição, `used` significa que uma instrução foi registrada sob a requisição e `reviewed` significa que alguém que não é o usuário leu o registro. Procure as duas saídas que não chegam a `reviewed`: um grant que permanece além do fim e uma revisão que nunca acontece.



Para que isso funcione, toda requisição feita durante um incidente carrega o identificador do incidente no motivo. O formulário do gateway pode exigí-lo, e uma requisição sem ele, em um incidente aberto, é recusada. Essa única convenção é o que permite a um job perguntar “tudo o que este incidente concedeu”.


```

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 não substitui a verificação de vencimento e a verificação de drift dos capítulos 8 e 3. Ele olha para os mesmos fatos do lado do incidente e dá ao incidente um gate: ninguém pode marcá-lo como encerrado enquanto um grant ainda estiver ativo ou um uso de break-glass não tiver sido revisado. Uma equipe que usa um sistema de tickets pode ligar `closeIncident` ao ticket, o que transforma “esquecemos de revogar” em um bloqueio visível.

Revisado

O último estado é o mais frequentemente pulado, porque o incidente parece ter acabado. A revisão é a que o Capítulo 8 descreve para o break-glass, estendida a todo grant que o incidente fez. Ela é sem culpa e pergunta sobre o sistema: o acesso era o menor que podia fazer o trabalho, e existia um caminho mais rápido que deveria ter existido? O Apêndice tem um modelo pós-incidente. O achado que compensa costuma ser um caminho que faltava.

Escritas durante um incidente

O reparo é o lugar mais tentador para relaxar um controle, porque o dano é visível e os dados estão errados agora mesmo. O Capítulo 10 estabeleceu que um reparo é uma correção de dados, e o Capítulo 8 estabeleceu que o break-glass não é uma rota para reparos. Sob pressão, manteria as duas decisões e tornaria rápido o caminho em conformidade, em vez de tornar disponível o caminho errado.

A primeira escolha é a qual caminho a necessidade pertence. A ordem das verificações vai da porta mais estreita à mais larga.

```
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)"
```

O break-glass só é acessível para a pequena ação limitadora: uma flag que impede um job de escrever linhas ruins, um status que segura um lote. O reparo duradouro dos dados é sempre uma correção de dados, com o mesmo raciocínio de antes: um reparo executado em uma sessão break-glass bruta pula o único controle que lê a instrução antes de ela ser executada, e roda onde um `UPDATE` sem limite ainda é possível (o Capítulo 8 nomeia isso como um risco aceito). Durante um incidente a tentação é maior, e também o custo de uma única escrita errada.

A correção de dados mantém sua forma. A requisição nomeia uma correção revisada e carrega um `approved_hash` tirado do registro de merge, o runner recusa um script cujo hash difira, o grant é derivado da correção, e a proteção por contagem e o snapshot continuam valendo (capítulos 9 e 10). O que o incidente muda é a revisão.

```
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
```

Isso espelha o lane expresso que o Capítulo 11 dá às migrações: mesmo job, mesma credencial, mesmo registro, tempo diferente. A revisão de acompanhamento é onde o atalho é pago de volta, e é também o que transforma um caminho acelerado em um evento contado. Se uma equipe descobre que a maioria das correções de um trimestre usa esse caminho, o lane normal é lento demais para o trabalho, e a resposta certa é acelerar o lane normal, não parar de contar.

Aviso: Uma correção que “precisa” de break-glass é um sinal sobre outra coisa, como diz o Capítulo 10: um teto que não tem uma coluna, um grupo de aprovadores pequeno demais para a hora ou uma coluna Restricted que pertence ao caminho de produto. Em um incidente, esse sinal é fácil de ignorar e caro de ignorar duas vezes.

Os casos que os capítulos anteriores adiaram

Três situações foram repassadas por capítulos anteriores. A tabela é a decisão de antemão; as subseções dão o raciocínio.

Situação	O que para	O que não muda	Decidido com antecedência
O gateway está fora do ar	O acesso humano aos dados (não há segunda porta)	Os serviços continuam rodando com as próprias credenciais	Quem restaura o gateway, e quando o instrumento selado pode ser usado

Situação	O que para	O que não muda	Decidido com antecedência
Uma restauração é necessária durante o incidente	Nada, se o caminho foi ensaiado	Aprovador, destino isolado, entrada no registro, exclusão agendada	Um aprovador nomeado, um destino preparado, um runbook de restauração
As aprovações são lentas à noite	Um reparo que não é emergência	A autoaprovação continua recusada	Uma ordem de escalonamento e quem pode ampliar o grupo

O gateway está fora do ar

Os capítulos 7 e 8 decidiram que não existe uma segunda porta: os humanos chegam ao banco de dados apenas pelo gateway, então uma indisponibilidade do gateway é também uma indisponibilidade do break-glass. O raciocínio ali era que uma credencial que funciona sem o gateway também funciona sem o logging, a identidade e a expiração. Manteria isso na pior das noites, e o primeiro movimento em uma indisponibilidade é notar o que não parou. Os serviços se conectam com as próprias credenciais e continuam rodando, então muitos incidentes em que o gateway por acaso está fora do ar não precisam de uma pessoa no banco de dados. Reverter o deploy, pausar o job, virar a feature flag, escalar o serviço: nada disso passa pelo gateway.

Se o incidente realmente precisa de acesso aos dados, o incidente agora são dois incidentes, e o primeiro é o gateway. Restaurá-lo vem antes da questão dos dados, e quem o restaura é a equipe de plataforma, usando o caminho de deploy e a redundância que o Capítulo 7 pediu.

Se todas as instâncias estão fora, não existe nenhum caminho humano aos dados, e o único instrumento que resta é o superusuário que o Capítulo 3 tratou como selado. É uma ferramenta de nível de motor, não um caminho de dados: ela contorna as proteções append-only e Restricted, e sua rota de rede é fechada para as pessoas. Limitaria seu uso a restaurar o gateway ou aquilo de que o gateway depende, nunca a ler ou escrever dados de negócio, e escreveria a regra.

Aviso: Não use o instrumento selado para olhar dados porque o gateway está fora do ar. Ele contorna as proteções append-only e Restricted, e o acesso que ele dá é o acesso que todos os capítulos anteriores foram construídos para impedir. Se os dados precisam ser lidos, restaure o gateway primeiro.

O uso sob a regra tem um mecanismo. A recuperação da credencial deixa um registro, uma segunda pessoa está presente e nomeada, abrir a rota de rede é uma mudança com responsável e hora de término, e a credencial é rotacionada depois porque uma pessoa a viu. O gateway não está registrando durante essa janela, então o log do motor é o único rastro. Depois que o gateway volta, a equipe de plataforma acrescenta uma entrada manualmente, como a entrada de auditoria externa que o Capítulo 14 descreve para as migrações, e o uso recebe a revisão que todo uso de break-glass recebe.

Uma restauração durante um incidente

O Capítulo 13 fez da restauração uma requisição: um aprovador que não é o solicitante, um destino isolado, uma entrada no registro, uma exclusão agendada e um início fechado. O custo nomeado ali era a velocidade durante um incidente, e a resposta era ensaiar e providenciar a aprovação com antecedência. Esse é o mecanismo inteiro, então vale explicitar o que “com antecedência” significa.

Antes do incidente, o runbook nomeia quem pode aprovar uma restauração à noite, o destino está preparado e isolado, e a requisição de restauração tem um formulário que aceita um identificador de incidente. Durante o incidente, a requisição segue sem alterações: o aprovador não é o solicitante, a entrada no registro vem primeiro, a exclusão agendada é definida antes que a cópia exista, e os logins são desativados e a definição de acesso é aplicada antes que a cópia seja aberta. O incidente acrescenta o motivo, não um atalho.

Para que serve a cópia importa mais do que como ela é feita. Durante um incidente, uma cópia restaurada é uma fonte de leitura, geralmente de valores antigos: o estado de uma coluna antes de uma escrita errada, ou linhas removidas por engano. Seus leitores são as pessoas da requisição. O reparo que se segue roda contra o primary como uma correção de dados,

com os valores lidos da cópia como entrada de um script revisado, e o `reapplyDeletions` do Capítulo 13 roda antes de a cópia ser aberta, para que a cópia não traga de volta dados que foram excluídos de propósito. Substituir o `primary` por uma cópia restaurada é uma decisão de recuperação, e pertence à equipe de plataforma e ao comandante do incidente. A questão de acesso dentro dela é a de cima: quem lê a cópia e quando ela é excluída.

Aprovações lentas à noite

O cenário noturno do Capítulo 8 terminou em uma decisão que este capítulo mantém: se o grupo de aprovadores não responde e a necessidade pode esperar, deixe uma nota e pare. Esse é um bom resultado, e um que a antiga `Ledgerly` não conseguia produzir. Para uma necessidade que não pode esperar, o escalonamento é embutido na requisição, de modo que um page sem resposta desce pelo grupo dentro da mesma requisição.

```
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
```

Duas regras cercam isso. Primeiro, o grupo só é ampliado por uma pessoa responsável por ele, e a ampliação é um registro: durante um incidente declarado, o comandante do incidente pode acrescentar um aprovador nomeado para o incidente, com um fim. Segundo, uma aprovação automática para quem estiver de on-call continua não sendo construída, porque um controle que concede toda requisição em segundos não é controle. Se as

aprovações regularmente ultrapassam a janela do grant, o achado é sobre a escala, o tamanho do grupo ou o horário, e o remédio é acrescentar pessoas ao grupo, não tirar o aprovador do circuito.

Ensaizando os caminhos

Um caminho que nunca rodou não foi testado. O capítulo inteiro é construído a partir de passos que ninguém quer descobrir pela primeira vez durante um incidente, então cada um precisa de um ensaio, chamado aqui de game day: um exercício agendado com um cenário, um resultado esperado e uma verificação sobre o registro. Quatro cenários cobrem o que este capítulo construiu. Eles são defensivos por design, cada um parte de um gatilho que o monitoramento já sabe enxergar, e cada um carrega uma condição de aprovação observável.

Cenário	Gatilho	Comportamento esperado	Condição de aprovação	Partes exercitadas
Vazamento de credencial	A credencial de um principal de simulação é declarada exposta	<code>containCompromise</code> em ordem; consulta de escopo anexada ao incidente	Nenhum login ativo para o principal, unfinished vazio, o escopo nomeia toda tabela que a simulação tocou	Identidade, Grant, Registro
Gateway fora do ar	Uma e depois todas as instâncias do gateway paradas em uma cópia de não produção	O acesso humano para; os serviços continuam rodando; o caminho de restauração é executado	A equipe de plataforma restaura o gateway usando o runbook; instrumento selado não usado, ou usado sob a regra dele com um registro	Caminho, Registro

Cenário	Gatilho	Comportamento esperado	Condição de aprovação	Partes exercitadas
Export suspeito	Um alerta de <code>rows_returned</code> alto, vindo de <code>evaluateAlertRules</code> (Capítulo 14)	Conter o principal; encontrar o destino no registro de cópias; excluí-lo	A cópia é encontrada, encerrada e registrada; a consulta de escopo concorda com o registro	Identidade, Registro, Execução
UPDATE em massa equivocado	Uma escrita que cabe no row cap e altera as linhas erradas, em um ambiente de ensaio	Interromper novas escritas; calcular a reversão a partir do snapshot; executá-la como uma correção de dados acelerada	Reversão aprovada por um revisor que não é o autor; revisão de acompanhamento enfileirada	Execução, Grant, Registro

Um game day só é aprovado pelo resultado observado, nunca por alguém lembrar o que fazer. Se uma simulação termina porque alguém conhecia um atalho, ela falhou, e o atalho passa a ser a coisa a construir. Anote os tempos que cada passo realmente levou; esses números pertencem à sua empresa, e não daria um número que não seja assim.

Use uma cópia de não produção sempre que possível, e principais sintéticos nas simulações em produção, para que nenhum dado real de cliente seja lido só para praticar a leitura. As cópias de simulação são registradas e encerradas como qualquer outra cópia (Capítulo 13).

Trade-offs e custos

O primeiro custo é o próprio ensaio. Os game days tiram engenheiros do trabalho de produto, e uma simulação precisa de um responsável e de um calendário. Para uma equipe de três pessoas, uma única simulação por ano cobrindo o vazamento de credencial é um começo razoável, e os outros três cenários podem esperar até a equipe crescer.

O segundo custo é a indisponibilidade que a contenção causa. Revogar a credencial de um serviço imediatamente é uma indisponibilidade nesse serviço, e conter `ledgerly_access_admin` interrompe o provisionamento. Ambas são escolhas a fazer com antecedência, com as pessoas que as suportam, porque quem decide às 3 da manhã não deveria estar ao mesmo tempo aprendendo o que a escolha custa.

O terceiro custo é a convenção do identificador de incidente. Ela acrescenta um campo a toda requisição e mais um motivo para um formulário ser recusado, e só é tão boa quanto a disposição das pessoas em preenchê-lo. A reconciliação que depende dela falha em silêncio quando o campo não é preenchido, então o gateway deve impor o campo e não confiar no hábito.

O quarto custo é a honestidade sobre o escopo: alguns incidentes terminam com um limite maior que a verdade, e relataria o limite em vez de alegar uma precisão que o registro não tem.

Não aplique toda a engrenagem a um sistema que ainda não tem um log de auditoria. `blastRadius` precisa do registro, e `reconcileIncidentAccess` precisa dos registros de requisições; uma equipe sem eles deve construí-los primeiro (capítulos 8 e 14) e, nesse meio-tempo, escrever um runbook curto que nomeie quem revoga, quem encerra sessões e quem é responsável pela pergunta sobre o que foi tocado.

Erros comuns

- **Rotacionar sem revogar.** Uma rotação muda o secret e deixa no lugar a sessão aberta e os privilégios concedidos. É tentadora porque a rotação é uma rotina que a equipe já tem. Revogue primeiro, depois encerre as sessões, depois rotacione.
- **Delimitar antes de conter.** A vontade de entender vem antes da vontade de parar, e uma equipe pode gastar uma hora na consulta enquanto o acesso continua ativo. Contenha primeiro. O registro sobrevive à contenção porque vive em um armazenamento que o acesso não consegue tocar.
- **Confiar no log sem verificá-lo.** Uma consulta de escopo sobre uma janela com uma lacuna parece uma resposta limpa. Verifique a cadeia e a sequência primeiro, e relate um limite inferior quando qualquer uma falhar.

- **Break-glass como lane de reparo.** Ele é rápido e está disponível, e não tem revisão de instruções. Use-o para a pequena ação limitadora e execute o reparo como uma correção de dados acelerada.
- **Deixar o acesso do incidente aberto.** O incidente parece acabado quando a causa é corrigida, e os grants ficam. Vincule o encerramento do incidente à reconciliação, para que o ticket não possa ser fechado enquanto houver acesso aberto.

Pontos-chave

- A urgência muda a velocidade de um controle, nunca a identidade que age, a instrução que é executada nem o registro que fica.
- A contenção tem uma ordem: revogar grants, encerrar sessões, rotacionar credenciais, delimitar o dano a partir do log de auditoria e, por fim, registrar e notificar, e todo passo roda mesmo que o anterior tenha falhado.
- Revogação não é rotação: uma rotação interrompe novos logins e nada faz pelos que já estão abertos, e é por isso que uma credencial estática vazada é cortada primeiro e substituída depois.
- O raio de impacto vem de uma única consulta sobre `access_audit_record`, e é um limite inferior sempre que a cadeia ou a sequência tem uma lacuna.
- O acesso em incidentes passa por requested, granted, used, revoked e reviewed, e um incidente não pode ser encerrado enquanto algum grant que ele criou ainda estiver aberto.
- Os reparos durante um incidente são correções de dados com uma revisão acelerada, e o break-glass fica para a pequena ação que interrompe o dano.
- Um caminho que não foi ensaiado é uma suposição, então o vazamento de credencial, a indisponibilidade do gateway, o export suspeito e o `UPDATE` em massa equivocado ganham cada um um game day com uma condição de aprovação observável.

O Capítulo 16 reúne os controles do livro inteiro em um checklist e uma escada de maturidade, e volta à frase a que todos os capítulos serviram.

Capítulo 16 — Um checklist de acesso e reflexões finais

Imagine a Ledgerly um ano depois do questionário de segurança do Capítulo 1. Outro cliente em potencial envia as mesmas duas linhas: “Liste todas as pessoas e sistemas que podem se conectar ao seu banco de dados de produção” e “Como vocês sabem o que eles fizeram?”. Alex abre o inventário de acessos em vez do gerenciador de senhas, lê a lista de portas e a devolve em uma tarde. Nada na resposta impressiona. Ela é curta, cada linha tem um responsável e um fim, e nada exigiu escavação.

Robin, que está de on-call nesta semana, pergunta pelo chat: “Temos um gateway, um runner, um pipeline, um armazenamento de auditoria e três jobs de revisão. De quais nós precisávamos, e quais construímos porque o livro mandou?”. Alex não tem uma resposta arrumada. Algumas peças provaram seu valor na primeira vez que um grant expirou sozinho; outras nunca foram usadas em situação real, e Alex não sabe dizer se isso significa que funcionam ou que estão paradas.

Este capítulo serve aos três pilares ao mesmo tempo: autorização, execução e responsabilização. Ele não acrescenta nenhum controle novo. Acrescenta as etapas do estado-alvo na ordem em que uma equipe consegue alcançá-las, os padrões pontuados lado a lado e um checklist cujas linhas são observações, não opiniões. A pergunta que ele responde é a que Robin fez: como uma equipe decide o que construir a seguir, e como sabe que um controle funciona?

O problema e por que ele é difícil

Quinze capítulos deixam quem lê com muitos controles e nenhuma ordem. Cada um foi apresentado com seu próprio raciocínio, e cada raciocínio era sólido no seu capítulo. Lidos em conjunto, parecem um projeto de um ano ou mais, e uma equipe diante dessa lista tende a fazer uma de duas coisas. Começa pelo que for mais barato e para quando o trabalho se

torna político, ou começa pelo que mais assusta e constrói uma peça elaborada do design enquanto a senha compartilhada ainda está em cinco arquivos de configuração.

A dificuldade é que “pronto” é difícil de enxergar no trabalho de acesso. Um esquema de papéis que existe no papel e um que é aplicado parecem iguais em uma reunião de revisão, então as equipes medem pelo que foi instalado, e instalado é a unidade errada. A unidade útil é um teste que alguém consegue executar: uma instrução que se consegue rastrear, uma conexão que é recusada, um grant que não existe mais.

Também é uma questão de como o sistema é construído, e não do que a política diz. Uma política afirma o que deveria acontecer; um design determina o que pode acontecer e o que deixa registro quando acontece. Cada linha abaixo pode ser respondida olhando para um sistema, um log ou uma tabela, e uma linha que só pode ser respondida perguntando a alguém o que pretende fazer não entra na lista.

A tese, concretizada

O livro defendeu uma frase, e é aqui que ela é conferida contra os capítulos:

O acesso a bancos de dados em produção não é um problema de distribuição de credenciais; é um problema de autorização, execução e responsabilização.

A frase tem três pilares, e cada pilar se apoia em capítulos específicos. Autorização é Grant mais Caminho, decidida por requisição. Execução é o que acontece com cada instrução. Responsabilização é Identidade mais Registro. Uma senha toca apenas o primeiro deles, e apenas na porta.

Pilar	Partes do modelo	Capítulos que o constroem	A pergunta que quem lê agora consegue responder
Autorização	Grant, Caminho	3, 5, 6, 8, 12	O que este principal pode fazer, onde, por quanto tempo e com quais colunas ocultas?
Execução	Execução	7, 9, 10, 11, 13	Como esta instrução chegou aos dados, quem ou o que a inspecionou e o que a limitou?

Pilar	Partes do modelo	Capítulos que o constroem	A pergunta que quem lê agora consegue responder
Respon-sabiliza-ção	Identidade, Registro	4, 7, 14, 15	Qual pessoa ou instância de workload agiu, por requisição de quem, e que evidência liga essa ação ao resultado?

Os capítulos 1 e 2 enquadram os três, e o Capítulo 15 os coloca à prova sob pressão. O Capítulo 7 aparece duas vezes porque o access gateway é ao mesmo tempo o caminho e o lugar em que uma pessoa se torna um login de sessão que um log consegue nomear; o Capítulo 13 fica sob execução porque a etapa de export do `reporting-job` executa um conjunto de instruções, e as outras portas laterais são rotas que contornam os mesmos controles.

Por que o trabalho com as credenciais era necessário e não era a resposta

O Capítulo 4 trata de credenciais, e é o capítulo que a maioria dos leitores espera que seja o centro do livro. Ele é necessário. Um serviço que lê sua senha de um arquivo compartilhado não tem identidade própria, e sem identidades distintas o registro não consegue nomear ninguém. Credenciais de curta duração e dinâmicas fazem da revogação uma etapa de rotina. A workload identity elimina o arquivo.

Nada disso decide o que um principal pode fazer depois de entrar, como uma instrução é verificada ou o que fica registrado. Uma equipe poderia concluir o Capítulo 4 com perfeição e ainda ter um papel que é proprietário de todas as tabelas, uma sessão de laptop que faz commit de cada instrução exatamente como foi digitada, e um log em que cada linha tem um nome distinto e nenhuma tem um motivo. Um cofre de senhas fornece os nomes distintos de que a parte Identidade precisa e, por meio da revogação, um pedaço do Grant; uma senha toca apenas o primeiro pilar, e apenas na porta. As outras partes são construídas em outro lugar, e o restante do livro mostra onde.

Quando uma proposta disser “vamos corrigir o acesso melhorando X”, situe X no modelo antes de aprovar:

```
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
```

A função não é um gate. Ela nomeia quais partes uma proposta deixa como estavam, porque uma parte vazia é um achado, e não uma omissão.

Um catálogo de padrões, pontuados por pilar

O Capítulo 1 traçou a gama de formas de entrar, e o Capítulo 7 comparou os caminhos concretos. A tabela abaixo coloca lado a lado os cinco padrões dessa gama e pontua cada um em relação aos três pilares. As pontuações são um julgamento meu, não uma medição. Elas dizem quanto de um pilar o padrão consegue evidenciar por si só, mantendo iguais os grants e o logging dos capítulos anteriores. Leia “forte” como “o padrão foi feito para comprovar isso”, “parcial” como “ele comprova parte disso, ou só se algo mais for acrescentado”, e “nenhum” como “o padrão é mudo aqui”.

Padrão	Autorização	Execução	Responsabilização
Acesso direto (cliente ao banco de dados)	Parcial: apenas grants, o alcance é o que a rede permitir	Nenhuma	Parcial, no máximo: logins por pessoa se alguém os provisionar, caso contrário nenhuma
Bastion	Parcial: estreita o caminho de rede	Nenhuma: ele encaminha, não distingue um SELECT de um DROP	Parcial: quem fez login no host, não o que foi executado
Session proxy ou access gateway	Forte: logins por sessão, expiração, grants por requisição	Parcial: limitada por grants e guarded settings, as instruções são digitadas livremente	Forte quanto a quem e quando, parcial quanto ao quê

Padrão	Autorização	Execução	Responsabilização
Executor que medeia as instruções	Forte	Forte: inspecionada antes de executar, vinculada a um approved hash	Forte: instrução, aprovador e resultado em um único registro
Caminho de produto (sem SQL)	Forte: a operação é a permissão	Forte: a operação é fixa	Parcial: a pessoa está na trilha de auditoria da ferramenta, o log do banco de dados mostra o login do serviço

Acesso direto

- **Comprova:** que um login existe e que seus grants eram os que eram.
- **Não comprova:** qual pessoa usou um login compartilhado, nem o que uma instrução fez.
- **Custo:** nenhum para construir, e é isso que atrai. Todo custo chega depois, na forma de uma semana de escavação.
- **Modo de falha:** a senha de admin compartilhada do Capítulo 1.
- **Quando não usar:** quando mais de uma pessoa ou workload consegue alcançar o login. É tolerável para um sistema de uma só pessoa cujo único consumidor é seu responsável, e escreveria o que faria isso deixar de ser verdade.

Bastion

- **Comprova:** que uma conexão passou por um único host controlado e, se as contas nele forem por pessoa, quem fez login ali.
- **Não comprova:** qual era qualquer instrução, já que o host não fala o protocolo do banco de dados.
- **Custo:** um host para aplicar patches e manter disponível, em troca de um único lugar para desligar o caminho de rede.
- **Modo de falha:** uma conta de bastion compartilhada transforma o host de novo na senha compartilhada, um salto antes.
- **Quando não usar:** quando você precisa de evidência no nível da instrução. Ele continua útil como alcance de rede na frente de algo que registra.

Session proxy ou access gateway

- **Comprova:** que uma pessoa identificada manteve um login de sessão específico, entre dois timestamps, sob grants específicos.
- **Não comprova:** que uma instrução digitada nessa sessão foi uma boa ideia. A sessão é um session handout (Capítulo 9) e qualquer coisa que os grants permitam pode ser digitada.
- **Custo:** um sistema novo que precisa estar disponível quando alguém precisar dele, e o caminho de aprovação que o acompanha (capítulos 7 e 8).
- **Modo de falha:** um gateway com uma segunda porta ao lado, que é uma senha compartilhada com um nome melhor.
- **Quando não usar:** quando as tarefas se repetem e têm um formato fixo. Elas pertencem a um caminho de produto, em que ninguém precisa de uma sessão.

Executor que medeia as instruções

- **Comprova:** que este conjunto exato de instruções, identificado por seu hash, foi aprovado por uma pessoa identificada, foi inspecionado, foi executado dentro de um limite e produziu um resultado contado.
- **Não comprova:** que a análise entendeu a instrução. A análise de consultas reduz erros e não é a fronteira de autorização, que continua com os grants (Capítulo 9).
- **Custo:** toda tarefa espera por revisão, nada é interativo, e alguém precisa construir e manter o runner.
- **Modo de falha:** uma aprovação que é sempre concedida em segundos, o que é teatro de segurança com etapas extras.
- **Quando não usar:** quando o trabalho é exploratório. Quem está tentando entender um estado precisa primeiro de uma sessão de leitura, e é para isso que serve `human_readonly`.

Caminho de produto

- **Comprova:** que uma operação definida foi executada, para um papel em uma ferramenta, com a trilha de auditoria da própria ferramenta.
- **Não comprova:** nada sobre as tarefas para as quais ninguém construiu uma funcionalidade. Perguntas livres não têm caminho.

- **Custo:** uma funcionalidade por tarefa, e o ciclo de mudança mais lento dos cinco.
- **Modo de falha:** o caminho de produto que acumula uma válvula de escape de “executar qualquer SQL” porque o primeiro caso urgente não tinha funcionalidade.
- **Quando não usar:** quando a tarefa ainda não se repetiu. O Capítulo 10 descreve o sinal para transformar uma correção repetida em funcionalidade, e uma tarefa que aconteceu uma vez não é esse sinal.

O padrão mais à direita nem sempre é o melhor, e um executor que medeia as instruções é mais do que uma pergunta sobre as faturas de um cliente exige. Escolheria por tarefa, com o padrão mais barato que ainda evidencie os pilares que a tarefa coloca em risco:

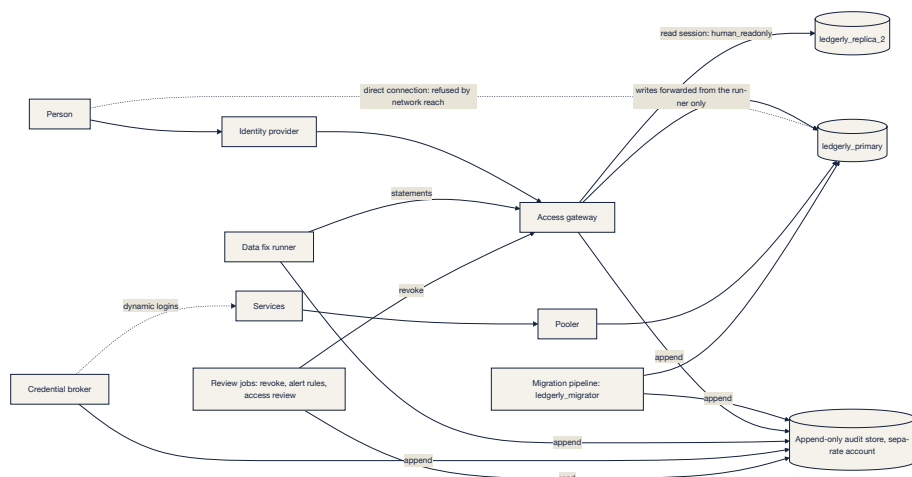
```
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
    (Chapter 11)
    if task.changesSchema:
        return "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"
```

A última linha é o ponto principal. Uma rota que não se encaixa entra no inventário com um responsável e uma data de término, como pede o bloco de inventário do Capítulo 1, e uma exceção sem os dois é um resquício.

A arquitetura de referência

O diagrama mostra o estado-alvo da Ledgerly com todos os controles do livro em uma só imagem. Observe três coisas: que nenhuma seta humana chega a um banco de dados a não ser pelo gateway, que toda escrita feita

por uma pessoa passa pelo runner, e que o armazenamento de auditoria fica em uma conta na qual os administradores de banco de dados não podem escrever.



A imagem simplifica: as réplicas de leitura das aplicações do Capítulo 6 dividem uma caixa com seus escritores, e a carga de analytics mascarada, o export para o fornecedor e os backups (Capítulo 13) ficam de fora, porque cada um é uma cópia com um responsável e uma expiração, e um desenho com todas as cópias não teria um centro legível.

Cada caixa corresponde a uma parte do modelo: o identity provider e o broker fornecem Identidade; os papéis, os logins de sessão e o job de expiração fornecem Grant; a regra de rede por trás da seta pontilhada e o gateway fornecem Caminho; o runner e o pipeline fornecem Execução; o armazenamento de auditoria e os jobs de revisão fornecem Registro. Uma caixa que não fornece nenhuma parte é candidata à remoção, e a pergunta de Robin merece esse teste.

Uma escada de maturidade com testes de saída

O estado-alvo é uma direção, não uma linha de chegada, então os passos do livro são agrupados aqui em cinco etapas. O agrupamento é meu. Ele segue quanto uma equipe consegue mudar antes que a mudança seguinte fique mais fácil, e organizações diferentes o reordenarão; a única ordem que defenderia é que o inventário vem primeiro, porque sem ele ninguém

sabe o que cada passo posterior precisa cobrir. Também é melhor começar cedo a manter registros, mesmo em forma simples, já que cada etapa posterior é mais fácil de avaliar com evidência.

Uma etapa está concluída quando seu teste de saída passa, e “nós instalamos” não é um teste. Cada teste é uma observação que alguém consegue repetir.

Etapas	O que acrescenta	Teste de saída	Por que as equipes estag-nam aqui
1. Conhecer as portas	Inventário de principais, portas, cópias (capítulos 1, 2, 13)	Um documento lista cada porta com um responsável e um fim, e uma pessoa recém-chegada à engenharia consegue responder “quem alcança o primary?” só com ele	O inventário é tratado como um projeto, então nunca termina e nunca recebe confiança
2. Nomear cada principal	Login e papel próprios por principal, credenciais dinâmicas, orçamentos por papel (capítulos 3, 4, 5)	Um dia do log do banco de dados não contém nenhum login compartilhado, e nenhum serviço se conecta como proprietário ou como admin	A virada parece um risco de indisponibilidade para cada serviço, então a senha compartilhada fica “por enquanto”
3. Um caminho, limitado	Gateway, réplicas, JIT e break-glass, classificação e mascaramento (capítulos 6, 7, 8, 12)	Uma conexão direta ao primary a partir de um laptop falha, um grant JIT desaparece em <code>expires_at</code> sem que ninguém aja, e uma sessão de leitura não retorna nenhuma coluna Restricted	Medo do gateway como ponto único de falha, e aprovadores que não conseguem responder à noite
4. Mediada e registrada	Política de execução, runner, correções revisadas, migrações somente pelo pipeline (capítulos 9, 10, 11)	Uma escrita feita pelo cliente da própria pessoa é recusada, e o runner recusa um conjunto de instruções cujo hash difere de <code>approved_hash</code>	A revisão vira o gargalo, então as exceções se acumulam mais rápido que as funcionalidades de produto
5. Comprovável e ensaiada	Armazenamento de auditoria com cadeia de hashes, regras de alerta, revisão de acessos, game days (capítulos 14, 15)	Uma instrução sem aviso, escolhida de ontem, pode ser rastreada até uma pessoa, uma requisição e um resultado sem perguntar a ninguém; <code>verifyChain</code> passa; um game day termina com o acesso revogado confirmado a partir do registro	O log existe e ninguém o lê, e o game day é a primeira coisa a ser cortada quando uma release atrasa

Uma etapa pode falhar no seu teste depois de ter sido declarada concluída, porque a configuração se desvia; um teste reexecutado periodicamente mostra o desvio enquanto ele é pequeno.

Nota: Uma equipe pequena, com um banco de dados e quatro pessoas na engenharia, poderia razoavelmente parar na etapa três e escrever o que faria a escada continuar. A disciplina está na razão escrita, não na altura.

O checklist

O checklist está agrupado por pilar, e cada linha é uma observação. A convenção para usá-lo é a mesma dos testes de saída: um “não” é um achado com um responsável e uma data, não uma falha a ser explicada e esquecida.

```
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
```

Autorização

- ☐ Toda tabela pertence a `ledgerly_owner`, e nenhuma pessoa ou serviço possui o login dele.
- ☐ Cada serviço se conecta com um login que é membro do seu próprio papel `app_*`, e nenhum papel de aplicação detém DDL.
- ☐ Não existe acesso de escrita permanente para nenhuma pessoa; `human_readonly` é o padrão e `human_jit_write` é um teto, não um papel concedido.
- ☐ Todo grant JIT e de break-glass tem um `expires_at`, e tanto o job de revogação quanto sua verificação independente de atrasos executaram recentemente.

- ☐ Uma conexão direta ao primary a partir de um laptop é recusada, e as réplicas rejeitam o login de admin.
- ☐ Colunas Restricted não aparecem em nenhuma requisição JIT, em nenhuma sessão de leitura e em nenhuma cópia de analytics, e o catálogo de classificação lista todas as colunas que existem.
- ☐ Todo backup, dump, export e cópia de staging tem um responsável identificado e uma expiração.

Execução

- ☐ As instruções de pessoas e de automação chegam aos dados pela política de execução, e a matriz deny / gate / allow está em controle de versão com um responsável.
- ☐ As sessões de escrita executam com auto-commit desligado e um row cap, e os guarded settings não podem ser alterados de dentro de uma sessão.
- ☐ Uma escrita JIT cita uma correção revisada, e o runner recusa qualquer conjunto de instruções cujo hash difira de `approved_hash`.
- ☐ Mudanças de vínculo de proprietário e escritas em colunas Restricted vão para o caminho de produto em todos os lanes.
- ☐ Mudanças de esquema executam como `ledgerly_migrator` a partir do pipeline, e nenhuma pessoa possui esse login.
- ☐ A etapa de export executa com um papel restrito, e seu arquivo tem um responsável identificado.

Responsabilização

- ☐ Cada login corresponde a exatamente uma pessoa ou uma instância de workload, e o log do banco de dados mostra isso.
- ☐ Todo registro de escrita e de sessão elevada traz a requisição, o aprovador, o fingerprint e as linhas retornadas, e os literais passam por redaction.
- ☐ O armazenamento de auditoria fica em uma conta na qual os administradores de banco de dados e `ledgerly_access_admin` não podem escrever, e `verifyChain` passa.
- ☐ Uma lacuna de sequência, uma sessão de gateway sem sessão de banco de dados, ou o inverso, gera um achado.

- ☐ `evaluateAlertRules` e `runAccessReview` executaram no último ciclo, e as revogações que propuseram foram aplicadas ou aceitas com uma razão.
- ☐ `containCompromise` foi ensaiada em um game day, e a consulta de raio de impacto foi respondida a partir do registro.

Cultura: o caminho seguro precisa ser o fácil

Todo controle do livro é um custo para quem o usa, e essa pessoa decide, na pior noite da semana, se vai pagá-lo. Um controle lento ou frequentemente fora do ar é contornado porque o design tornou a rota segura a difícil, e um incidente não espera. A resposta útil é tratar o desvio como um defeito do design.

Colocaria uma regra no centro da cultura e a tornaria mensurável:

```
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
```

Dois hábitos sustentam a regra. Conte com que frequência o papel de break-glass é declarado e pergunte, a cada vez, se um caminho JIT ou uma funcionalidade de produto deveria ter atendido no lugar; uma contagem em alta diz que o caminho normal é lento demais, e uma contagem zero durante um ano pode significar que o papel funciona ou que ninguém acredita que ele será permitido, o que um game day resolve. E dê aos aprovadores algo que eles consigam responder à noite: uma aprovação que leva horas ensina os solicitantes a pedir mais do que precisam, e uma que é sempre concedida em segundos ensina todo mundo a não lê-la (o Capítulo 8 trata do checklist do aprovador).

Trade-offs e custos

Um capítulo de encerramento que apenas recomenda repetiria o erro contra o qual o livro alerta, então aqui está o que o design inteiro custa. O sistema da arquitetura de referência tem muitas partes, e todas elas precisam estar disponíveis no momento em que alguém precisar. O gateway, o runner, o pipeline, o broker, o armazenamento de auditoria e os jobs de revisão têm, cada um, um responsável, um caminho de on-call e um modo de falha. Uma equipe de plataforma pequena pode perceber que gasta seu tempo mantendo os controles funcionando, e esse é um custo real a pesar contra a exposição que eles removem.

O checklist e a escada têm custos próprios. Um checklist lido a cada trimestre vira um ritual a menos que os achados sejam encerrados, e uma escada vira um relatório de status quando as equipes otimizam pelo nome da etapa; o teste de saída é o antídoto, porque é barato de reexecutar e difícil de forjar. Uma pontuação “forte” diz para que um padrão foi feito, não que uma determinada implantação dele cumpra isso.

Quando não usar o aparato deste capítulo: um sistema de uma só pessoa, com um banco de dados e sem fornecedor, precisa de um login de proprietário que ninguém usa, um papel somente leitura, um backup que restaura e um inventário de uma página. O livro é para o ponto em que o número de principals ou o custo de um erro ultrapassa isso, e não sei onde fica esse ponto na sua empresa; ele depende de quantas pessoas conseguem alcançar os dados, de quanto deles é Restricted e de quanto tempo um erro permanece invisível.

Erros comuns

- **Tratar o checklist como um certificado.** É tentador terminar uma rodada e arquivar o resultado. O checklist mostra onde falta evidência no dia em que foi lido. Reexecute-o periodicamente e mantenha os achados abertos até que cada um seja resolvido ou aceito com uma razão e uma data.
- **Construir primeiro o padrão mais à direita.** Um executor que medeia as instruções é atraente porque parece o maior controle. Sem o inventário e sem principals distintos por baixo, ele medeia as instru-

ções de um login compartilhado. Suba a escada em ordem, ou escreva por que uma ordem diferente serve melhor para você.

- **Acrescentar uma segunda porta para emergências.** Parece resiliência e reconstrói a senha compartilhada. Se uma equipe precisar de uma enquanto o gateway amadurece, ela precisa de um responsável, de um alerta no uso e de uma condição de desativação (Capítulo 8).
- **Ler um desvio como um problema de conduta.** Quando alguém contorna um controle, o design tornou isso mais barato que o caminho pretendido. Pergunte quanto o caminho pretendido custava naquele momento e mude isso.

Pontos-chave

- O acesso a bancos de dados em produção é um problema de autorização, execução e responsabilização, e uma senha toca apenas o primeiro deles, e apenas na porta.
- O trabalho com credenciais do Capítulo 4 é necessário porque a identidade torna o registro possível, e não é suficiente porque nada diz sobre o que um principal pode fazer ou sobre o que uma instrução faz.
- Escolha entre acesso direto, um bastion, um access gateway, um executor que medeia as instruções e um caminho de produto por tarefa, com o padrão mais barato que ainda evidencie os pilares que a tarefa coloca em risco.
- Uma etapa de maturidade está concluída quando seu teste de saída passa, e um teste que alguém consegue reexecutar é a única forma de “pronto” que sobrevive ao desvio.
- Cada linha do checklist é uma observação de um sistema, de um log ou de uma tabela; um “não” é um achado com um responsável e uma data, nunca um veredito sobre uma pessoa.
- O caminho seguro precisa ser o fácil, e todo desvio é informação sobre onde o design tornou a rota pretendida custosa.
- Os controles reduzem o raio de impacto e tornam o comportamento visível, e custam disponibilidade, atenção e espera, então nomeie esses custos antes de adotá-los e de novo quando forem pagos.

Alex responde à pergunta de Robin na manhã seguinte com um teste, e não com uma defesa: uma peça fica se fornece uma parte do modelo que nada mais fornece e se o seu teste de saída passa. O questionário voltará, de outro cliente ou de outro auditor, e as mesmas duas linhas estarão nele. O que o livro pode deixar para você é um sistema que responde à primeira linha a partir de um documento, responde à segunda a partir de um registro, e deixa quem está de on-call fazer a coisa certa sem um esforço heroico.

Apêndice

Este apêndice é material de referência para os três pilares a partir dos quais o livro argumenta: autorização, execução e responsabilização. Ele reúne seis partes: um glossário, uma referência rápida dos padrões pontuados no Capítulo 16, a matriz deny / gate / allow do Capítulo 9, uma correspondência entre perguntas de auditores e artefatos, e dois modelos, um para a revisão de acessos do Capítulo 14 e outro para a revisão pós-incidente do Capítulo 15.

Nada aqui é novo. Quando uma tabela repete uma de um capítulo, as células são as do próprio capítulo, e se as duas diferirem, o capítulo é a fonte. A interpretação de qualquer regime cabe ao seu jurídico e à sua área de conformidade; a correspondência lista artefatos de engenharia e não afirma mais do que isso.

A. Glossário

Os termos estão listados em ordem alfabética, com o capítulo que constrói cada um. Esta é a referência final do livro para o seu vocabulário: cada entrada é a forma curta, e o capítulo citado contém o relato completo.

Termo	Significado
access gateway	Um intermediário que reconhece identidades, autentica uma pessoa, emite uma credencial de curta duração e registra a sessão. Não é um bastion, e não é um proxy de banco de dados (Capítulo 7).
access_audit_record	As entradas gravadas para cada instrução, uma do tipo intent e outra do tipo outcome, contendo o principal, o login, a requisição, o <code>fingerprint</code> da instrução, as tabelas e colunas tocadas e, no outcome, as contagens de linhas e o resultado. É a parte Registro do modelo (Capítulo 14).
acesso just-in-time (JIT)	Acesso concedido para uma tarefa específica e por um tempo limitado (Capítulo 8).
acesso permanente	Acesso que existe continuamente, seja ele usado ou não (Capítulo 1).

Termo	Significado
allowlist, denylist	Uma lista do que é permitido e uma lista do que é recusado. Nunca “white-list” nem “blacklist” (Capítulo 7).
análise de consultas	Inspecionar uma instrução antes de executá-la: seu tipo, as tabelas e colunas que ela toca, se uma escrita tem uma guarda de linhas delimitada e a contagem de linhas esperada. Reduz erros e não é a fronteira de autorização (Capítulo 9).
approved hash	O <code>approved_hash</code> de uma requisição de acesso: o hash do conjunto de instruções aprovado, tirado do registro de merge da correção revisada ou do registro da requisição aprovada, nunca da entrada do runner (capítulos 9, 10).
aprovação vinculada à instrução	Uma aprovação anexada ao conjunto exato de instruções que será executado, identificado por <code>approved_hash</code> , em vez de apenas a uma janela de tempo ou a um conjunto de privilégios (capítulos 8, 9, 10).
bastion	Um host endurecido que é o único caminho de rede até um sistema protegido. Ele encaminha tráfego e não entende o protocolo do banco de dados (Capítulo 7).
break-glass	Acesso de emergência com logging pesado e revisão obrigatória. O papel é <code>human_breakglass</code> , e a requisição é autodeclarada (Capítulo 8).
cadeia de hashes	Um formato de registro em que cada entrada carrega o hash da anterior, de modo que uma entrada alterada ou removida quebra o <code>verifyChain</code> . Fornece evidência de adulteração e não a impede (Capítulo 14).
caminho de produto	Uma operação definida em uma ferramenta, como o <code>admin-console</code> pedindo ao <code>payments-service</code> que altere uma coluna <code>Restricted</code> , sem SQL para a pessoa (capítulos 9, 10).
classes de dados	As quatro classes nomeadas de dados: Restricted (a exposição é difícil de desfazer, como <code>iban</code> e <code>token_value</code>), Confidential (dados pessoais ou comerciais sobre clientes e contatos), Internal (dados operacionais sem conteúdo pessoal), Public (seguros para publicar). Escritas com inicial maiúscula quando usadas como nomes de classe (capítulos 2, 12).
classificação de dados	Separar os dados nas classes nomeadas que decidem as regras de tratamento (Capítulo 12).
consulta ad hoc	Uma consulta que uma pessoa escreve e executa uma vez, fora de qualquer aplicação (Capítulo 10).
contenção	Interromper o acesso de um principal em uma ordem fixa: revogar grants, encerrar sessões, rotacionar credenciais, delimitar o dano, registrar e notificar (Capítulo 15).
cópia de analytics	O armazenamento <code>ledgerly_analytics</code> . É uma cópia dos dados de produção, não uma réplica (Capítulo 6).

Termo	Significado
cópia de dados	Qualquer lugar em que os dados de produção foram duplicados: réplicas, a cópia de analytics, backups, dumps, exports, dados de staging (capítulos 6, 13).
correção de dados	Uma alteração nos dados de produção feita por uma pessoa fora do caminho normal da aplicação, para reparar um estado que a aplicação não consegue reparar (Capítulo 10).
credencial	O que um principal apresenta para se autenticar: uma senha, um certificado, um token. Uma credencial é um tipo de secret (Capítulo 4).
credencial de curta duração	Uma credencial que expira sozinha em um prazo curto e definido (Capítulo 4).
credencial dinâmica	Uma credencial de curta duração gerada sob demanda para um único principal, de modo que duas requisições nunca recebem o mesmo valor (Capítulo 4).
credencial estática	Uma credencial que permanece a mesma até que alguém a altere (Capítulo 4).
deny / gate / allow	As respostas da política de execução, sempre nesta ordem. Deny nunca executa por este caminho. Gate só executa após aprovação adicional, ou só pelo caminho de produto. Allow executa dentro dos grants da sessão (Capítulo 9).
execução	Como uma instrução chega aos dados: quem ou o que a submete, se algo a inspeciona antes de executar, o que limita seu efeito e como ela sofre commit (capítulos 1, 9).
execução mediada	Um modo em que o principal nunca detém uma conexão: ele submete uma instrução, o gateway ou um runner a analisa e o runner a executa (Capítulo 9).
four-eyes	Aprovação em que uma segunda pessoa revisa antes de uma ação arriscada ser executada. Diz quem aprova, não o que é aprovado (Capítulo 8).
grant	Dar um privilégio a um papel, um papel a um login ou a outro papel ou, apenas para JIT, um privilégio sobre colunas nomeadas diretamente a um login de sessão. Também o registro resultante (Capítulo 3).
grant em nível de coluna	Um grant de um privilégio sobre colunas nomeadas de uma tabela. Onde o motor não oferece isso, as colunas Restricted ficam em tabelas separadas fora do teto, e os grants JIT descem para o nível de tabela (capítulos 3, 8).
guarded settings	Configurações de sessão que um principal não pode alterar de dentro de uma sessão: statement timeout, row cap, modo de auto-commit, padrão somente leitura, papel. Só são chamadas de não sobrescrevíveis onde um mecanismo as impõe (capítulos 5, 9).
identidade	A afirmação verificada de que um principal é quem diz ser. Um papel não é uma identidade (Capítulo 1).

Termo	Significado
independente do motor	A expressão do livro para notação e conselhos que não pertencem a nenhum motor de banco de dados (Prefácio).
log de auditoria	O registro de quem fez o quê com quais dados, mantido onde o autor não pode alterá-lo. A tabela <code>audit_events</code> é o registro de negócio da aplicação, não o log de auditoria (Capítulo 14).
login	A conta, no nível do banco de dados, com a qual um principal se conecta (capítulos 1, 3).
maskamento	Substituir ou ofuscar valores de modo que o valor real não fique visível enquanto o formato continua utilizável. O maskamento estático altera uma cópia, o maskamento dinâmico altera o que uma consulta retorna (Capítulo 12).
migração	Uma mudança de esquema versionada, executada por <code>ledgerly_migrator</code> . “Mudança de esquema” é a ideia geral, “migração” é o artefato (Capítulo 11).
minimização	Não coletar, copiar nem manter dados de que ninguém precisa (Capítulo 12).
mudança de vínculo de proprietário	Um <code>UPDATE</code> de uma coluna que liga uma linha ao cliente proprietário dela, como <code>customer_id</code> em <code>bank_accounts</code> . Passa por gate e é encaminhada ao caminho de produto (Capítulo 9).
papel	Um conjunto nomeado de privilégios no banco de dados. Um papel nunca é uma pessoa (Capítulo 3).
PII	Personally identifiable information. “Dados sensíveis” é o termo guarda-chuva para os dados Restricted e Confidential (Capítulo 2).
porta lateral	Uma cópia de dados ou um caminho de export que contorna os controles de acesso do primary (capítulos 6, 13).
primary, réplica	O banco de dados gravável e suas cópias de leitura. A cópia de analytics não é uma réplica (Capítulo 6).
principal	Qualquer coisa que se autentica: uma pessoa, um serviço, um job, uma integração de fornecedor (Capítulo 1).
principal de automação	Um principal que não é uma pessoa: um job, um runbook, um comando de ChatOps, um agente de IA. Tem uma workload identity e um padrão somente leitura, e o registro nomeia tanto ele quanto a pessoa que o acionou (Capítulo 9).
privilégio	Uma coisa específica que um grant permite, como <code>SELECT</code> em uma tabela (Capítulo 1).

Termo	Significado
privilégio mínimo	Conceder apenas o que uma tarefa precisa, pelo tempo de que ela precisa, contando o custo de fazê-lo (capítulos 1, 3).
proxy de banco de dados	Um intermediário que entende o protocolo e que faz pooling, roteia ou filtra conexões de banco de dados. É chamado de pooler quando o assunto é apenas pooling (capítulos 5, 7).
raio de impacto	Quanto pode ser afetado se um principal, uma credencial ou uma ação der errado. Sempre declarado a respeito de uma coisa específica (capítulos 1, 2).
redaction	Remover um valor por completo de uma saída ou de um log (Capítulo 12).
registro	A parte do modelo que pergunta que evidência permanece e se ela liga a instrução à pessoa, à requisição e ao resultado (Capítulo 14).
revisão de acessos	Uma verificação agendada de que o acesso de cada principal ainda é necessário. Compara o que foi concedido com o que foi usado, e uma pessoa revisora nomeada decide (Capítulo 14).
revogação, rotação	A revogação faz uma credencial ou um grant parar de funcionar agora. A rotação substitui uma credencial em um calendário ou por suspeita, e não encerra as sessões que já estão abertas (capítulos 4, 15).
runbook	Um procedimento escrito e seguido para uma tarefa repetível (Capítulo 10).
runner	O runner de correção de dados revisado. Ele executa um conjunto de instruções somente se o seu hash for igual a <code>approved_hash</code> , e o gateway encaminha escritas JIT apenas a partir dele (Capítulo 10).
secret	Qualquer valor que concede acesso se for divulgado (Capítulo 4).
sessão	Uma conexão autenticada, ou uma interação intermediada pelo gateway, de um principal (capítulos 5, 7).
session handout	Um modo em que o gateway entrega a um principal um login de sessão e uma conexão, e o próprio cliente do principal envia as instruções. Os grants do login limitam tudo (Capítulo 9).
teatro de segurança	Um controle que acrescenta atrito sem reduzir a exposição (capítulos 2, 8).
teto	O máximo que uma requisição JIT pode pedir, definido por <code>human_jit_write</code> . Nunca uma coluna Restricted (capítulos 3, 8).
tokenização	Substituir um valor por um substituto que apenas um serviço controlado consegue mapear de volta. Não é nem criptografia nem mascaramento (Capítulo 12).
workload identity	Uma identidade que a plataforma dá a um workload em execução para que ele consiga se autenticar sem uma senha armazenada (Capítulo 4).

Vários desses pares são confundidos, e uma regra curta resolve cada um antes que comece uma discussão.

Se a pergunta é	Use	Não
Um host que encaminha tráfego de rede, sem conhecimento das instruções	bastion	access gateway
Um intermediário que emite logins de sessão por pessoa e registra as sessões	access gateway	proxy de banco de dados
Fazer pooling ou rotear conexões, nada sobre pessoas	pooler, ou proxy de banco de dados	access gateway
O cliente da própria pessoa envia instruções por uma conexão que o gateway entregou	session handout	execução mediada
A pessoa submete uma instrução e nunca detém uma conexão	execução mediada	session handout
Fazer uma credencial parar de funcionar agora	revogação	rotação
Quem aprova uma ação arriscada	four-eyes	aprovação vinculada à instrução
O que exatamente foi aprovado, por hash	aprovação vinculada à instrução	four-eyes
O registro de negócio da aplicação	audit_events	log de auditoria

B. Referência rápida de padrões

O Capítulo 16 pontua cinco padrões em relação aos três pilares. As pontuações são o julgamento do livro e não uma medição: “forte” significa que o padrão foi feito para comprovar o pilar, “parcial” que ele comprova parte disso ou só com algo acrescentado, “nenhuma” que ele é mudo. A coluna de custo é a que o Capítulo 16 apresenta, abreviada.

Padrão	O que resolve	Custo	Autorização	Execução	Responsabilização	Capítulos
Acesso direto (cliente ao banco de dados)	Um login existe e seus grants eram os que eram	Nenhum para construir; todo custo chega depois, na forma de uma semana de escavação	Parcial: apenas grants, o alcance é o que a rede permitir	Nenhuma	Parcial, no máximo: logins por pessoa se alguém os provisionar, caso contrário nenhuma	1

Padrão	O que resolve	Custo	Autorização	Execução	Responsabilização	Capítulos
Bastion	Um host controlado, um lugar para desligar o caminho de rede	Um host para aplicar patches e manter disponível	Parcial: estreita o caminho de rede	Nenhuma: ele encaminha, não distingue um SELECT de um DROP	Parcial: quem fez login no host, não o que foi executado	7
Session proxy ou access gateway	Uma pessoa identificada manteve um login de sessão específico, entre dois timestamps, sob grants específicos	Um sistema novo que precisa estar disponível quando alguém precisar dele, e o caminho de aprovação que o acompanha	Forte: logs por sessão, expiração, grants por requisição	Parcial: limitada por grants e guarded settings, as instruções são digitadas livremente	Forte quanto a quem e quando, parcial quanto ao quê	7, 8
Executor que medeia as instruções	Este conjunto exato de instruções, por hash, foi aprovado, inspecionado, limitado e contado	Toda tarefa esperada por revisão, nada é interativo, alguém constrói e mantém o runner	Forte	Forte: inspecionada antes de executar, vinculada a um approved hash	Forte: instrução, aprovador e resultado em um único registro	9, 10
Caminho de produto (sem SQL)	Uma operação definida foi executada, para um papel em uma ferramenta	Uma funcionalidade por tarefa, e o ciclo de mudança mais lento dos cinco	Forte: a operação é a permissão	Forte: a operação é fixa	Parcial: a pessoa está na trilha de auditoria da ferramenta, o log do banco de dados mostra o login do serviço	9, 10

Escolher entre eles é uma decisão por tarefa, e a regra é a que o Capítulo 16 apresenta: o padrão mais barato que ainda evidencie os pilares que a tarefa coloca em risco.


```
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 operações perigosas

A matriz abaixo é a política de execução do Capítulo 9, repetida com suas células inalteradas. As linhas são as classes de instrução que o classificador produz. As colunas são os lanes em que uma instrução pode chegar. Cada célula traz uma de três respostas: deny (nunca executa por este caminho), gate (precisa de uma aprovação que ainda não tem, ou pertence a outra rota), allow (executa dentro dos limites do loop e, onde a célula diz “bound”, apenas dentro do limite da fonte citada).

Classe de instrução	Sessão de leitura humana	Escrita JIT	Break-glass	Runner de correção de dados	Migração
Leitura, dentro do row cap	Allow	Allow	Allow	Allow	Allow
Escrita delimitada, dentro da contagem aprovada	Deny	Allow, vinculada à correção (somente runner)	Allow, vinculada a um row cap (nota obrigatória)	Allow, vinculada ao approved hash e à contagem	Allow, vinculada à regra de backfill
Escrita sem delimitação	Deny	Deny	Deny	Deny	Gate: aprovador nomeado

Classe de instrução	Sessão de leitura humana	Escrita JIT	Break-glass	Runner de correção de dados	Migração
Mudança de vínculo de proprietário	Deny	Gate: caminho de produto	Gate: caminho de produto	Gate: caminho de produto	Gate: aprovador nomeado
Escrita em uma coluna Restricted	Deny	Gate: caminho de produto	Gate: caminho de produto	Gate: caminho de produto	Gate: aprovador nomeado
Export	Gate: por tier	Deny	Deny	Deny	Gate: aprovador nomeado
Mudança de esquema (DDL)	Deny	Deny	Deny	Deny	Allow: lane revisado
Mudança de privilégio	Deny	Deny	Deny	Deny	Gate: revisão da plataforma
Escrita em <code>ledger_entries</code> ou <code>audit_events</code>	Deny	Deny	Deny	Deny	Gate: revisão da plataforma

Três regras de leitura poupam a maioria das consultas.

- **Uma escrita nas mãos de uma pessoa é negada ou limitada.** A sessão de leitura humana não tem caminho de escrita, as escritas JIT vêm apenas do runner, e o break-glass é o único lane em que uma pessoa escreve diretamente, sob um row cap e com uma nota.
- **Um gate nomeia para onde a instrução vai em seguida.** “Caminho de produto” significa que o `admin-console` pede ao `payments-service`, uma segunda pessoa aprova e a mudança é registrada em `audit_events`. “Aprovador nomeado” e “revisão da plataforma” significam que o lane de migração sinaliza a instrução para uma pessoa que precisa dizer sim.
- **A matriz não é a fronteira de autorização.** A análise de consultas reduz erros. Os grants do login continuam sendo a fronteira, e uma ins-

trução que o classificador interpreta mal ainda é limitada por eles (Capítulo 9).

A decisão para uma instrução, escrita como regra:

```
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. Correspondência com requisitos de conformidade

Os regimes de proteção de dados e de cartões de pagamento tendem a fazer uma família de perguntas sobre acesso. Um auditor que trabalha sob um desses regimes normalmente vai querer ver que um controle existe, que ele executou e que seus resultados foram examinados. Esta correspondência lista os tipos de pergunta, o controle que responde a cada um, o artefato que ele deixa e o capítulo que o constrói. É a tabela do Capítulo 14, repetida, e não afirma que nenhum controle satisfaça nenhuma obrigação.

Pergunta	Controle	Artefato	Capítulo
Quem pode alcançar os dados de produção, e por qual caminho?	Gateway como única porta, esquema de papéis como código, verificação de desvio	Lista de regras de rede, definições de papéis, relatório de desvio	3, 7
Quem executou uma determinada instrução, com quais grants e qual aprovação?	access_audit_record com id da requisição e aprovador	O registro e o relatório de verifyChain	14

Pergunta	Controle	Artefato	Ca- pí- tulo
Quanto tempo dura o acesso elevado, e quem o aprovou?	Requisição JIT com aprovador e <code>expires_at</code> , job de revogação	Registros de requisições, log de revogações, verificação de grants em atraso	8
Como os dados sensíveis são identificados e limitados?	Classificação no esquema, mascaramento, colunas retidas	O catálogo, a verificação da carga mascarada	12
Onde estão as cópias dos dados de produção, e quem é responsável por cada uma?	Registro com responsável e data de término, <code>sweeper</code>	Registro, log do <code>sweeper</code>	6, 13
Quem altera o esquema, e quem revisou?	Pipeline de migração, branch protegida, entrada de início externa	Registros de migração conferidos com commits e entradas de auditoria	11
O acesso ainda é necessário?	Revisão de acessos com revisores nomeados	Revisão concluída com decisões	14
O que acontece em uma emergência?	Break-glass com alerta e revisão obrigatória	Registros de revisão, histórico de alertas	8, 15
Quem administra o próprio acesso?	Logins de <code>ledgerly_access_admin</code> como produtores separados	Seus streams e a comparação entre catálogo e registro	3, 14

Um controle que existe mas nunca executou, ou que executou e nunca foi examinado, responde a apenas um terço da pergunta. A verificação abaixo é a que se deve executar antes de entregar um artefato a quem o pediu.

```
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. Modelo de revisão de acessos

Este é o modelo por trás do `runAccessReview` do Capítulo 14. Copie-o a cada ciclo de revisão. A revisão propõe e uma pessoa revisora nomeada decide; ninguém revisa o próprio acesso, e o período é mais longo que o ciclo mais longo do workload revisado (uma revisão de um mês proporia revogar um grant que um relatório trimestral usa em dois meses de cada três).

Cabeçalho

```
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 em comparação com o uso

Grant (papel, privilégio, tabela) ou login	Responsável	Último uso no registro	Achado	Decisão	Decidido por	Concluído em
<role>, <privilege>, <table>	<named principal>	<date> or none>	sem uso / sem responsável / uso mais restrito que o grant / correto	manter / restringir a <privilege> / revogar / atribuir responsável	<reviewer>	<date>

Associe o uso por privilégio e não apenas por tabela. Um papel que faz select toda noite e nunca fez update é candidato a ser restringido de S, U para S. Inclua os mapeamentos de grupos do identity provider para papéis e os logins de `ledgerly_access_admin`.

Parte 2: coisas que não são grants

Item	Pergunta	Resposta	Responsável	Decisão
Registro de cópias de dados (Capítulo 13)	Cada cópia ainda tem um responsável e uma data de término futura?	<yes / no, which>	<name>	<keep / end by date>

Item	Pergunta	Resposta	Responsável	Decisão
Quem pode aprovar requisições JIT	Cada aprovador ainda está em um papel que o justifica?	<yes / no, which>	<name>	<keep / remove>
Quem pode ler o armazenamento de auditoria	Cada pessoa leitora ainda precisa disso?	<yes / no, which>	<name>	<keep / remove>
Principais de automação (jobs, runbooks, agentes de IA)	Cada um ainda executa, com uma workload identity e um padrão somente leitura?	<yes / no, which>	<name>	<keep / retire>

Parte 3: usos de break-glass no período

Id da requisição	Usado por	Incidente ou razão	Revisado por	Data da revisão	Achado sobre o sistema
<id>	<principal>	<reason>	<other than user>	<date or MISSING>	<for example: missing path>

Uma linha sem data de revisão é, por si só, um achado.

Regras de encerramento

```
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. Modelo de revisão pós-incidente

Este modelo segue o Capítulo 15. Ele é sem culpa: cada pergunta é sobre o sistema que permitiu uma ação ou a tornou invisível, e nenhuma pergunta quem deveria ter sabido melhor. Preencha-o a partir do registro, não da

memória. Onde o registro não consegue responder a uma pergunta, escreva isso, porque a lacuna é um achado. Os horários são os que o registro mostra; não estime.

1. Resumo

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. Contenção, em ordem

Passo	Concluído em	Resultado	Notas sobre o que atrapalhou
Revogar grants	<timestamp>	feito / falhou	
Encerrar sessões	<timestamp>	feito / falhou	
Rotacionar credenciais	<timestamp>	feito / falhou	
Delimitar o dano a partir do log de auditoria	<timestamp>	feito / falhou	
Registrar e notificar	<timestamp>	feito / falhou	

Perguntas: Cada passo foi executado mesmo que um anterior tenha falhado? Algum passo ficou inacabado, e foi acompanhado? O log estava acessível e, se não, as entradas ficaram em fila sem atrasar a remoção do acesso?

3. Raio 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 do acesso durante o incidente

Id da requisição	Tipo	Solicita-do	Concedi-do	Primeiro uso	Revoga-do	Revisado por
<id>	jit / breakglass	<time>	<time>	<time or none>	<time>	<other than user, or MISSING>

Perguntas: Algum grant permaneceu além de `expires_at`? Algum acesso foi concedido sem que nenhuma requisição o explicasse? O `closeIncident` encontrou acesso aberto, e ele foi encerrado antes do incidente?

5. Escritas durante o incidente

Escrita	Rota (runner com revisão acelerada, caminho de produto, break-glass direto)	Revisor que não era o autor	Approved hash conferido
<description>	<route>	<name>	sim / não

Pergunta: O break-glass direto foi usado para uma escrita em que o runner teria atendido, e por quê?

6. Casos adiados, se ocorreram

- **Gateway indisponível:** por quanto tempo o acesso humano ficou parado, se os serviços continuaram em execução e se o runbook de restauração funcionou como escrito.
- **Restauração durante o incidente:** se a cópia restaurada foi registrada com um responsável e uma data de término, e encerrada.
- **Aprovações lentas à noite:** qual requisição esperou, por quem, e o que a regra para uma requisição sem resposta fez.

7. Achados e acompanhamento

Achado sobre o sistema	Parte (Identidade, Grant, Caminho, Execução, Registro)	Mudan-ça	Respon-sável	Termina em
<for example: a path that should have existed>	<part>	<change>	<name>	<date>

Encerre com uma verificação: todo achado é uma mudança em um design com um responsável e uma data de término, e não um lembrete para ter mais cuidado?

SOBRE O LIVRO

O acesso a bancos de dados em produção não é um problema de distribuição de credenciais; é um problema de autorização, execução e responsabilização.

Um cofre de senhas melhor muda apenas a forma como um secret chega a uma pessoa. Ele não decide o que essa pessoa pode fazer, o que na verdade é executado sobre os dados nem quem responde por isso depois. Este livro acompanha a Ledgerly de uma senha de admin compartilhada até um modelo de acesso projetado de propósito: identidades com privilégio mínimo, acesso just-in-time e break-glass, execução mediada para operações perigosas e uma trilha de auditoria capaz de dizer quem fez o quê e por quê. Os padrões são independentes do motor, e cada um vem com o custo que carrega.

SOBRE O AUTOR

Muhammet Şafak desenvolve software desde 2008. Durante muitos anos atuou ativamente nas áreas de saúde, finanças e telecomunicações, e defende que o desenvolvimento deve se concentrar no problema, não na tecnologia. Desde 2024 projeta e desenvolve sistemas de alto tráfego escaláveis de ponta a ponta.

Faz mentoria de equipes de desenvolvimento de software de pequeno e médio porte, conduz projetos de código aberto e registra por escrito as decisões, com suas razões e seu custo.

Sistemas confiáveis não são construídos com peças confiáveis. São construídos em torno das peças que falham.

I
ACESSO

II
EVENTOS

III
AGENTES

