공유하는 시스템

한 플랫폼에 수많은 고객을 쓰는 멀티테넌트 설계

ThakiCloud (Hyojung Han)

공유하는 시스템

한 플랫폼에 수많은 고객을 쓰는 멀티테넌트 설계

ThakiCloud (Hyojung Han)

Contents

1 1장. 한 서버에 많은 고객을 얹는 법: 멀티테넌트 설계 입문

당신은 하나의 제품을 가진다. 처음엔 당신 자신을 위해, 혹은 한 고객만을 위해 만들었다. 그런데 이제 세 명, 열 명, 서른 명의 고객이 같은 제품을 쓰고, 각자의 데이터가 같은 서버 한 대에 쌓인다.

이 책은 바로 그 상황을 어떻게 안전하게 돌리느냐에 대한 것이다. 한 대의 서버에서 여러 고객의 데이터를 함께 돌리되, 서로의 영역이 절대 뒤섞이지 않게 하는 설계. 그리고 그 설계 위에 테넌트 단위의 가격을 얹는 실무다.

1.1 테넌트가 뭔지

멀티테넌트 시스템에서 테넌트는 "고객"을 가리키는 단위다. 같은 애플리케이션 인스턴스를 쓰면서 서로의 데이터가 보이해서는 안 되는 사용자 집합. 한 회사 하나가 하나의 테넌트다. 디자인 스튜디오도 테넌트이고, 옆집 두 명 빵집도 테넌트다.

중요한 건, 테넌트가 사용자인 게 아니라 경계라는 점이다. 같은 테넌트 안에서는 사용자가 데이터를 공유해도 된다. 한 스튜디오의 대표와 팀장이 같은 문서를 보는 건 정상이다. 경계 너머로는 한 줄도 보이지 않아야 한다. 빵집이 스튜디오의 고객 계약서를 보는 건 허용되지 않는다.

이 구분 때문에, "사용자 계정별로 데이터를 나눠야지"라는 생각은 이미 틀린 출발이다. 사용자 계정은 경계가 아니다. 계정은 경계 안에서 움직이는 단위일 뿐이다. 로그인한 사용자가 누구든, 그 사용자가 속한 테넌트가 어디인지 먼저 정해져야 하고, 그 다음에 그 테넌트 안에서만 데이터를 볼 수 있다.

이 책의 모든 설계 결정은 한 질문으로 요약된다. 경계를 어디에 두고 누가 어떻게 지켜내느냐.

1.2 왜 1인 개발자가 이걸 생각해야 하는가

1인 개발자의 가장 큰 자산은 시간이고, 가장 큰 부채도 시간이다. 당신은 서른 개의 데이터베이스 인스턴스를 관리할 수 없다. 서른 개의 백업 잡을 돌릴 수도, 서른 번의 다른 업그레이드 티켓에 응답할 수도 없다. 서버가 하나면 배포는 한 번이고, 백업은 한 번이고, 보안 패치는 한 번이다. 고객이 열만 명이더라도.

그래서 고객이 늘면 자연스러운 본능은 "인프라는 하나, 다 같이 쓴다"는 것이다. 그게 멀티테넌트다. 한 대의 하드웨어 비용에 N명의 고객을 얹는 구조는, 1인 개발자가 SaaS로 갈 때 사실상 유일한 현실적인 경로다. 고객이 한 명 늘 때마다 서버를 한 대 더 사기 시작하면, 수익은 늘기 전에 운영이 먼저 무너진다.

근데 멀티테넌트에는 숨은 비용이 있다. 신뢰다. 고객이 당신에게 사는 건 기능이기도 하지만, 동시에 데이터를 맡기는 행위이기도 하다. 고객 A의 데이터가 고객 B한테 새어나가는 순간, 그건 버그가 아니라 계약 위반이다. 한 고객과의 관계만 깨지는 게 아니다. "이 플랫폼에선 내 데이터가 옆집한테도 보일 수 있다"는 사실이 알려지는 순간, 모든 고객이 같은 불안에 시달리게 된다.

대부분의 멀티테넌트 사고는 대수롭지 않은 실수에서 온다. WHERE 절을 하나 빼먹은 것, 테넌트를 구분하지 않는 검색 API, 테넌트가 없는 캐시 키, 기본값으로 모든 테넌트를 보여주는 관리자 API, 그리고 모든 사용자에게 돌려보내는 밤샘 이메일 잡. 공격자가 필요한 사고가 아니다. 평범한 금요일 저녁의 평범한 배포다.

그러니 멀티테넌트 설계의 진짜 목표는 성능이 아니라, 실수를 어렵게

만드는 일이다. 필터를 하나 잊으면 쿼리가 하나 깨지되, 사업 전체가 깨지지 않는 시스템. 사람의 기억에 의존하지 않고 시스템 자체에 경계를 박아넣는 설계.

1.3 세 가지 실패 모드

본격적으로 가기 전에, 흔히 빠지는 모양을 세 개 알아두자. 대부분의 멀티테넌트 사고는 이 중 하나로 시작된다.

1.3.1 격리 없는 상태

하나의 고객을 위해 앱을 만들고 두 번째 고객을 받으면서 아무것도 바꾸지 않은 경우. 테넌트 컬럼이 없다. 경계라는 개념이 없다. 이 단계에서는 모든 데이터가 "앱의 것"으로 섞여 있다.

문제를 깨닫는 시점은 보통 첫 고객이 "우리 데이터만 따로 볼 수 있나요?"라고 물었을 때다. 계약 검토 중일 수도 있고, 경쟁사가 그랬다고 해서일 수도 있다. 그런데 그때는 이미 모든 테이블이 뒤섞여 있다.

여기서 중요한 사실 하나. 격리는 일방통행이다. 데이터가 섞이기 전에 나누는 쪽이 압도적으로 쉽다. 이미 섞인 데이터를 정직하게 분리하려면, 각 줄이 누구의 것인지를 역추적하는 작업이 필요하고, 그 역추적은 당신에게만 전해졌던 입말에 의존하게 된다. 입말은 코드다. 하지만 코드처럼 테스트되지 않는다.

1.3.2 과격리

끝 반대편에는, 고객마다 데이터베이스를, 서버를 따로 만들어주는 경우가 있다. 다섯 명까지는 괜찮다. 쉰 명이면 마이그레이션 스크립트가 밤새 돌아가고, 이백 명이면 그냥 멈춰선다.

1인 개발자가 과격리로 기울어드리는 이유는, 격리가 유일한 안전으로 느껴지기 때문이다. "각자 따로"가 가장 마음 놓이는 구조다. 하지만 N개를 관리하는 것은 1인 개발자의 천적이다. 마이그레이션이 N배, 백업이 N배, 비밀번호가 N배, 그리고 어느 하나가 죽으면 N개의 고객 중 하나가 죽는다. 기술은 N을 줄이면서도 안전을 유지하는 것, 즉 어느 고객에게 더 많은 격리가 필요한지 알아내는 일이다.

1.3.3 절반만 한 격리

가장 흔하고 가장 위험하다. 데이터베이스는 테넌트로 나눴는데, 캐시는 그러지 못했다. 큐 메시지에 테넌트가 없고, 파일 저장소의 경로에도 테넌트가 없다.

DB에서는 테넌트가 교차하지 않더라도, 검색 결과에 옆집 파일명이 뜰 수 있다. A 테넌트의 키로 캐시에 넣은 값이 B 테넌트의 응답으로 돌아올 수 있다. 작업 큐에 들어간 메시지에서 테넌트가 빠지면, 워커가 "어떤 데이터베이스에 접속할지"를 알아낼 길이 없다.

절반 격리는 격리 없는 것보다 나쁘다. 없으면 적어도 경계가 없다는 걸 알므로, 한 번에 모두를 고칠 수 있다. 절반이면 안전하다고 착각한다. 그리고 새는 곳은 당신이 잊은 부분이다. 캐시, 큐, 파일, 검색. 데이터가 지나가는 모든 통로에 경계가 있어야 하는데, 통로는 늘 생각했던 것보다 많다.

1.4 이 책이 쓸 예시

구체적으로 만들겠다. 하나를 정해서 책을 관통시킨다.

제품은 "스튜디오". 소규모 스튜디오를 위한 문서와 사진 관리 서비스다. 테넌트는 이렇게 셋이다. 두 명 빵집, 여섯 명 디자인 스튜디오, 오십 명 에이전시. 셋 다 같은 사이트에 접속하고, 같은 코드를 쓰고, 같은 서버에

데이터를 저장한다.

다만 빵집은 디자인 스튜디오의 고객 계약서를 절대 봐선 안 된다. 그리고 에이전시가 API 한도를 다 써도, 빵집의 페이지는 같은 속도로 열려야 한다.

이 문장에 요구가 둘 들어 있다. "절대 못 본다"는 격리다. 2장의 주제. "같은 속도"는 쿼터와 noisy neighbor다. 3장과 4장의 주제. 5장은 이 위에 돈을 얹는 방법이다. 빵집은 작은 티어를 쓰고, 에이전시는 큰 티어를 쓰며, 둘 다 같은 인프라에서 돌아가는 것. 그게 이 책이 도달하는 지점이다.

1.5 프레임워크의 4층

멀티테넌트 설계는 "한 번의 아키텍처 결정"으로 생각하기 쉽지만, 실제로는 독립적인 4층이다. 각 층은 다른 수준에서 풀리며, 다른 비용을 가진다.

- 격리: 경계를 어디에 두나. tenant 컬럼, 스키마, 데이터베이스. (2장)
- 쿼터와 레이트 리밋: 테넌트가 얼마나 쓰게 될 것이냐. (3장)
- noisy neighbor: 한 테넌트가 다 먹어가면 어떻게 할 것이냐. (4장)
- 과금과 운영: 돈을 어떻게 받겠고, 무엇이 터졌을 땐 뭘 할 것이냐. (5장)

순서가 있다. 격리를 먼저 고른다. 데이터가 어디에, 어떻게 놓이느냐가 정해져야, 그 데이터에 얼마를 치겠는지 말할 수 있다. 그 다음에야 쿼터를 얼마로 정할지가 결정된다. 그리고 noisy neighbor 대응은 격리 방식에 제약을 받는다. 한 데이터베이스에 있을 때 쓰는 수법과, 한 테넌트 하나의 데이터베이스를 가질 때 쓰는 수법은 다르다.

한 층을 건너뛰어 시간을 아끼면, 보통 사고로 돌아온다. "일단 격리만 하고 한도는 나중에"라고 한 시스템에서 나중에 찾아오는 건 한도가 아니라, 한도 때문에 터진 사고다.

1.6 언제 이걸 안 하는가

솔직한 주의 하나. 만약 당신의 제품이 한 고객만 두고 있고, 앞으로도 그럴 것 같다면 멀티테넌트 시스템을 만들지 말라. 두 번째 고객이 영영 없을지도 모르는 제품에 테넌트 컬럼을 붙이는 것은 회수 없는 비용이다. 모든 쿼리에 필터를 달고, 모든 캐시 키에 접두사를 붙이고, 그걸 평생 지킬 필요는 없다.

멀티테넌트는 "두 번째 고객이 나타났다"는 순간을 위한 기술이다. 시작 신호는 구체적이다. 두 번째 유료 고객. 데이터 분리를 요구하는 고객. SLA를 언급하는 계약서. 이런 신호가 보이면, 이 책을 읽기 시작할 때가 된 것이다.

그 신호 이후, 이 책이 걷는 길은 짧다. 격리 모델을 고르고, 테넌트에게 계량을 붙이고, 한도를 정하고, 돈을 받게 만든다. 네 단계. 장이 하나씩 그 단계다.

2 2장. 격리의 형태: tenant 컬럼 vs DB 분리

1장은 격리를 "먼저 고르는 층"이라고 했다. 이 장은 그 고르는 법이다. 다만 한 가지를 먼저 짚겠다. 격리는 스펙트럼이다. "tenant 컬럼이냐 DB 분리냐"의 이분법은 아니다. 한 끝에서 다른 끝까지 이어지는 continuum이며 설계의 목표는 특정 지점에 박는 것이 아니라, 나중에 테넌트를 그 스펙트럼 위를 이동시킬 수 있게 여지를 남기는 것이다.

2.1 스펙트럼의 4단계

고객을 얼마나 물리적으로 떼느냐에 따라, "다같이 쓴다"에서 "완전 따로"까지 이렇게 나뉜다.

L1: tenant 컬럼. 모든 테넌트가 하나의 데이터베이스 안에 있고 테이블의 각 줄이 tenant_id라는 컬럼으로 식별된다.

L2: 스키마 분리. 하나의 데이터베이스 안에서, 테넌트마다 테이블의 이름 공간이 따로 있다.

L3: DB 분리. 하나의 데이터베이스 인스턴스 안에서, 테넌트마다 데이터베이스가 따로 있다.

L4: 완전 분리. 테넌트마다 데이터베이스 인스턴스가, 때로는 서버까지 따로 있다.

단계가 오를수록 관리가 단순해지는 대신 격리가 약해진다. L1은 관리할 대상이 하나라 가장 쉽지만 물리적 격리는 가장 약하다. L4는 격리가 가장 강하지만 관리 대상이 테넌트 수만큼 늘어난다. 1인 개발자는 보통 L1로 시작해서, 필요해지는 테넌트만 L3이나 L4로 올린다.

2.2 L1: tenant 컬럼, 기본값

L1에서 고객 데이터가 들어가는 모든 테이블에 tenant_id 컬럼이 붙는다. 사용자 테이블, 문서 테이블, 청구서 테이블, 전부.

2.2.1 테이블의 모양

기본키 선택이 미세하게 영향을 준다. (tenant_id, id)의 복합 기본키는 모든 쿼리가 본질적으로 테넌트 범위 안에 놓이게 한다. 가장 강한 형태다. 다만 id의 모양이 바뀌어서 단순 id를 전제로 하는 코드에서 소음이 생긴다.

더 흔히 쓰는 실무 형태는, 단순 id 기본키에 tenant_id가 맨 앞에 오는 인덱스를 붙이는 것이다. 이 경우 WHERE tenant_id = ?라는 필터가 당신을 보호하는데, 그래서 필터를 빼먹지 않게 하는 구조, 혹은 최소한 습관이 필요하다.

인덱스의 디테일은 안전보다 성능에 더 큰 영향을 준다. 테넌트로 필터하는데 인덱스가 created_at으로 시작하는 쿼리는 전체 테이블을 훑는다. "테넌트 범위 안"인 모든 인덱스는 tenant_id를 맨 앞에 두어야 한다.

2.2.2 강제하는 세 층

tenant 컬럼은 그것을 강제하는 코드만큼만 강하다. L1에는 강제하는 층이 셋 있다.

첫째, 애플리케이션 층. 요청이 시작되면 미들웨어가 이 요청이 어느 테넌트에 속하는지를 정해서, 요청 컨텍스트에 넣는다. 쿼리 라이브러리나 ORM이 그 컨텍스트 안의 모든 쿼리에 WHERE tenant_id = ?를 자동으로 붙인다. 핵심은, 필터를 선택이 아니라 default로 만드는 일이다. 필터를 직접 써야 한다면, 사람은 빼먹는다.

둘째, 데이터베이스 층. PostgreSQL에는 row-level security, RLS라는 것이 있다. 각 테이블에 "이 세션의 변수와 tenant_id가 일치하는 줄만 보인다"는 정책을 만들어 두고, 애플리케이션이 연결될 때 그 변수를 설정한다. RLS는 backstop이지, 애플리케이션 스코핑의 대체제가 아니다. ORM을 우회하는 버그가 지나가면, RLS가 잡는다. 그리고 세션 변수를 설정하지 못한 연결은 아무것도 보이지 않게 되는데, 이렇게 실패해도 안전한 방향이다.

셋째, 운영 층. 대부분의 사람이 빼먹는 부분이다. 배치 잡, cron, 관리 도구, 백업 스크립트. 이 경로들은 요청 미들웨어를 거치지 않는다. "매일 밤 모든 사용자에게 리포트를 보낸다"는 잡은 테넌트별로 루프를 돌리야 한다. 관리 대시보드의 "전체 보기" 화면은 따로, 감사되는, 특권을 가진 경로여야 한다. 요청 경로에는 테넌트 경계가 있는데 잡 경로에는 없으면, 새는 곳은 잡 경로다.

2.2.3 L1이 충분한 때

L1은 이런 조건에서 좋은 기본값이다. 테넌트가 작고 동질적일 때. 규제 요구를 가진 고객이 없을 때. 데이터 총량이 적당할 때. 500개 테넌트가 하나의 PostgreSQL을 공유하면서 각 테넌트가 작은 스튜디오라면, L1은 충분하고 충분한 동안 L1에 머물러 있어야 한다.

2.3 L3: DB 분리, 필요해지는 테넌트를 위해

L3에서 각 테넌트는 같은 인스턴스 안에서 자신만의 데이터베이스를 가진다. 애플리케이션은 tenant registry라는 작은 테이블을 갖고 tenant_id를 연결 문자열, 혹은 데이터베이스 이름으로 매핑한다. 요청이 시작되면 미들웨어가 테넌트를 정하고, 풀이 그 테넌트의 데이터베이스로 가는 연결을 건넨다.

L3가 사는 것.

- 물리적 분리. 코드 층의 습관에 의지하지 않는다. 테넌트를 넘는 쿼리는 “필터가 막아주는 것”이 아니라, 구조적으로 불가능하다.
- 깔끔한 삭제. 고객이 떠나면, 데이터베이스를 drop한다. “모든 테이블에서 모든 줄을 삭제한다”는 스크립트가 어떤 테이블을 빼먹을 위험이 없다.
- 테넌트별 한도. 그 테넌트의 연결 수를 cap하고 무거운 마이그레이션을 스케줄로 돌리고 그 하나만 스냅샷을 만든다.
- 팔 수 있는 이야기. “당신의 데이터는 별개의 데이터베이스에 있다”는, 기업 계약에 쓸 수 있는 문장이다.

그 대가.

- 마이그레이션이 배가된다. 모든 스키마 변경이 테넌트 데이터베이스마다 한 번씩 돈다. registry를 루프하고 데이터베이스마다 version을 추적하는 마이그레이션 러너가 필요하다.
- 풀 관리. 테넌트별 풀을 테넌트 수만큼 가지면, 유휴 연결이 쌓인다. cap과 공유 풀러가 없으면, 인스턴스를 소진한다.
- 코드 두 갈래. 대부분의 테넌트가 L1이고 일부가 L3이면, 데이터 접근 층이 격리 수준에 따라 갈라진다. 그 갈림은 영구적인 기술 부채다. 그래서 명확한 trigger를 갖고 그 안으로 들어가야 한다.

2.4 비교를 하나

구분 | tenant 컬럼 (L1) | DB 분리 (L3) |
격리 강도 | 앱과 RLS에 맡김 | 구조적으로 분리 |
관리 단수 | 데이터베이스 1개 | 테넌트 수만큼 |
삭제 방식 | 테이블 지분 정지 | 데이터베이스 drop |
대테넌트 대응 | 어려움 | 쉬움 |
초기 복잡도 | 낮음 | 중간 |

2.5 L2와 L4, 양 끝의 말

L2(스키마 분리)는 대부분의 사람이 건너뛰는 중간 옵션이다. 하나의 데이터베이스 안에서 테넌트별 이름 공간을 얻지만 마이그레이션 도구와 ORM 지원이 종종 어색하고 L1+RLS에 비해 격리가 얻는 게 운영 비용에 비해 작다. L2가 손에 걸린다면, 실제로 원하는 건 L3라는 신호인 경우가 많다.

L4(별개 인스턴스)는 규제 때문에 오는 경우다. 계약상 물리적으로 별개의 데이터베이스를 요구하는 고객, 혹은 데이터 거주 요건. 운영비가 비싸다. 1인 개발자는 계약서에서 명시적으로 요구할 때만 L4를 손에 대고, 그렇게 하면 가격을 그만큼 받아야 한다.

2.6 promotion path

살아남는 설계는 hybrid다. default는 L1이고 registry에 isolation_level라는 컬럼이 있고, 정해진 trigger가 테넌트를 L3로 올린다. trigger는 필요해지는 전에 적어 두어야 한다. 이런 식이다.

- 고객의 계약이 분리를 요구할 때. 규제, 기업 조항.
- 테넌트의 데이터가 임계값을 넘을 때. 예를 들어 공유 데이터베이스의 10% [추정]
- 테넌트가 noisy neighbor가 되어, L1의 한도만으로 담아내지 못할 때. (4장)
- 고객이 명시적으로 요구하면서 그 대가를 지불할 때.

promotion은 실무에서는 일방통행이다. L3에서 L1으로 되돌리는 건 데이터를 합치는 일이고 그걸 하고 싶을 때는 드물다. 그래서 promotion의 순간의 질문은 “이걸 할 수 있는가”가 아니라, “이 테넌트를 영원히 더 비싸게 돌리고 싶은가”다.

코드의 결과. 데이터 접근 층에 깔끔한 seam 하나가 필요하다. 테넌트 컨텍스트를 받아, 쿼리가 공유 테이블로 가는지 별개 데이터베이스로 가는지 숨겨주는 repository 인터페이스. 그 seam이 깔끔하면 L1과 L3가 같이 산다. 깔끔하지 못하면, 테넌트 체크가 핸들러 곳곳에 흩어져, 새로운 기능마다 동전 던지기가 된다.

2.7 기존 데이터에 테넌트를 붙일 때

대부분의 1인 개발자는 처음부터 L1로 시작하지 않는다. 한 고객을 위한 앱이 있고 두 번째 고객이 온다. 그때 데이터를 분리하는 순서가 있다.

- 컬럼부터. 모든 고객 데이터 테이블에 tenant_id를 추가한다. nullable로 시작해야 한다. 아직 데이터를 채우지 못했기 때문이다.
- 백필. 기존 데이터는 전부 첫 테넌트의 것이므로, tenant_id를 그 테넌트의 id로 채운다. 이 작업은 한 번만, 주의 깊게. 채워진 줄의 수와 전체 줄의 수가 일치하는지 확인한다.
- 강제. 애플리케이션 층의 스코핑을 켜고 RLS 정책을 배포한다. 그리고 기존 고객은 이제 “첫 테넌트”로, 시스템 안에서 경계 안으로 들어온다.
- 두 번째 테넌트. 이 단계가 끝나기 전까지는 두 번째 테넌트를 만들지 않는다. 백필이 끝나기 전에 두 번째 테넌트가 데이터 쓰기 시작하면, 그 줄에는 tenant_id가 없거나, 틀린 것이 된다. 순서가 전부다.

이 순서를 거치면, “한 고객용 앱”이 “두 고객용 플랫폼”이 된다. 그리고 이 장의 나머지가, 그 플랫폼을 열 명, 백 명으로 넓히는 법이다.

2.8 경계 체크리스트

어느 단계를 고르든, 경계는 데이터가 움직이는 모든 곳에 있어야 한다. 짧은 체크리스트.

테넌트는 인증된 identity에서 온다. 요청 body나 query parameter에서 오지 않는다. 자신의 tenant_id를 말할 수 있는 사용자는, 남의 tenant_id도 말할 수 있는 사용자다.

캐시 키에 테넌트가 들어간다. tenant:{id}:key, 혹은 테넌트별 캐시 이름 공간.

큐 메시지에 tenant_id를 실어 나른다. 워커는 데이터에 손 대기 전에 컨텍스트를 설정한다.

파일 저장소 경로는 테넌트별로 접두사가 있고 접근은 파일 id가 아니라 테넌트를 대조한다.

검색 인덱스는 테넌트 범위 안에 있다. global 인덱스도, 엔진 층에서 모든 쿼리가 테넌트로 필터하면 된다.

관리 경로는 별개이고 로그되고 default가 아니다. "모든 테넌트 보기"는 capability이지, shortcut이 아니다.

그리고 자기 자신을 갚아주는 테스트 하나. cross-tenant 속성 테스트. 두 테넌트를 만들고 한 테넌트에 대해 일단의 random 작업을 돌리고 다른 테넌트가 아무것도 보지 않는지를 assert한다. CI에서 돈다. 빼먹은 필터, 캐시 miss, 틀린 루프의 잡을 잡는다. 당신이 모든 쿼리를 올바르게 쓸 것을 기억하지는 못할 것이다. 이 테스트가 기억한다.

2.9 규모에 대해 한마디

흔한 걱정. "하나의 데이터베이스가 1,000개 테넌트를 감당할 수 있을까?" 답은 보통, "감당하지 못할 때까지, 그렇다"다. L1를 부수는 건 드물게 row count다. 대개는 무거운 테넌트다. 다른 999개 테넌트의 데이터와 쿼리 양을 압도하는 한 고객. 데이터베이스는 멀쩡하다. 그러나 p99 레이턴시와, 다른 999개 고객의 경험이 아니다. 그게 4장이 다룰 주제다.

3 3장. 쿼터와 레이트 리밋: 테넌트 한도를 정하는 실무

2장은 경계가 어디에 놓이는지 정했다. 이 장은 그 경계에 계량기를 달아, 테넌트가 쓸 수 있는 양을 숫자로 만든다. 메커니즘이 둘 있는데, 같은 것이 아니므로, 먼저 구분부터 한다.

3.1 쿼터와 레이트 리밋은 다른 동물

쿼터는 기간의 총량이다. "이 테넌트는 한 달에 API 호출 10만 건." 비즈니스의 질문에 답한다. 무엇을 파느냐, 그리고 그 양을 얼마로 정할 것이냐.

레이트 리밋은 throughput의 모양 제한이다. "아무리 해도 초당 10개는 넘지 못하게." 엔지니어링의 질문에 답한다. 한 테넌트가 다른 테넌트에 해를 끼치기 전에 얼마나 빠르게 밀어붙일 수 있느냐.

타임라인, 저장 방식, 강제 지점, 발동 시 의미가 각각 다르다. 한 테넌트는 한 달 내내 쿼터 안에 있으면서 짧은 burst에서 레이트 리밋에 닿을 수 있다. 반대로 하루 만에 월 쿼터를 다 쓰고 남은 일수는 한가하게 있을 수 있다. 둘을 혼동하면, 측정도 못 한 것에 돈을 청구하거나, 그냥 허용하려던 고객을 throttle하거나, 둘 중 하나를 하게 된다.

3.2 무엇을 측정할 것인가

청구할 수 있는 것은 측정하는 것에 한하고 측정할 수 있는 것은 counter를 정의한 것에 한한다. SaaS에서 의미가 있는 meter는 보통 몇 가지만 있다.

- 저장공간. 테넌트의 데이터가 차지하는 공간.

- 호출. API 요청, 혹은 그 일부. export, webhook, OCR 같은 특정 종류의 것.
- 좌석. 로그인할 수 있는 사람의 수.
- 배치 작업. 테넌트가 일으키는 background job.
- 출력. 시스템 밖으로 나가는 데이터의 양.

여기서 원리는 하나다. 가격을 고르기 전에 meter를 고른다. 3개월 차에 "export를 과금하고 싶은데, 그걸 counter하지 못했네"라고 깨달으면, 이미 추측으로 돌아간 것이다.

3.3 티어와 숫자

1인 개발자는 보통 두 티어로 시작하고 숫자는 자신의 비용 모델에서 나와야 한다. 다음 표는 모양을 구체적으로 만들기 위한 예시다.

항목 | Starter | Pro |
저장공간 | 1GB | 20GB |
월 API 호출 | 10만 | 100만 |
초당 요청 | 10 | 50 |
팀원 | 3 | 20 |

중요한 건 이 숫자가 아니다. 중요한 건, 각 행이 위 meter 목록의 meter이며 각 meter가 티어별 한도를 갖고 있다는 점이다. 새 meter가 생길 때, 예를 들어 OCR을 제공하기 시작할 때, 그 meter는 가격보다 먼저 모든 티어에 한 줄씩 들어간다.

3.4 레이트 리밋 알고리즘

흔한 알고리즘은 정밀도가 높아지는 순서대로 이렇다.

3.4.1 fixed window

counter와 TTL 하나. "초당 10개"가 "매초 reset되는 counter"가 된다. 단순하고 싸다. 그리고 seam이 있다. 0.9초를 기다리다가 20개를 쏘는 클라이언트는, 경계에서 20개를, 원래 한도의 두 배를 통과시킬 수 있다. 1인 개발자에게는, 이건 종종 받아들이기 어렵지 않다. burst의 크기는 정해져 있고 짧으니까.

3.4.2 sliding window

이동하는 window에서 요청 수를 정확히 추적한다. 더 정확하고 더 많은 state를 갖는다. fixed window의 seam은 사라지지만, memory와 logic으로 대가를 치른다.

3.4.3 token bucket

버킷이 capacity까지 토큰을 갖고 각 요청이 하나를 쓰고 토큰은 steady rate로 다시 찬다. capacity가 burst 허용량이고 refill rate가 steady 한도다. 실제로 원하는 것을 모델링한다. "잠깐은 빨리 가도 된다. 하지만 영원히 그런 건 안 된다." 테넌트별로, 버킷의 capacity와 refill은 티어에서 온다. burst가 중요한 것에선, 그리고 거의 모든 것에 그렇지만 이게 추천이다.

3.5 어디서 강제하고 멀티프로세스 함정

요청 안의 강제 순서는, 인증, 테넌트 해석, 레이트 리밋, 쿼터 체크, 그리고 handler다. 레이트 리밋이 쿼터보다 앞선 이유는, 레이트 리밋이 싸고 flood로부터 당신을 보호하고, 쿼터 체크는 더 느린 state를 건드리기 때문이다.

함정은 process count다. 앱이 3개 프로세스를 돌리는데, 각 프로세스가

10 rps로 설정된 local 레이트 리미터를 갖고 있으면, 테넌트는 실제로 30 rps를 받는다. workarounds가 몇 가지 있다.

- 나누기. 각 프로세스가 limit / N을 강제한다. 단순하고 조금 부족하고 N이 알려져 있고 일정해야 한다.
- 중앙화. Redis에 shared counter를 두고 요청당 INCR 한 번. 정확하지만 요청당 round trip이 발생하고, Redis가 hot path 위에 놓인다.
- 두 층. 나누어진 값의 local 리미터에, aggregate를 잡는 굵은 global counter를 더한다. 가장 정확하고 가장 많은 코드다.

1인 개발자에게는, 나누기가 보통 첫 move로 맞다. 오차는 N 안에 갇히고 N은 작고 테넌트가 불평하기 시작하면 나중에 중앙화로 옮길 수 있다.

3.6 발동했을 때 무엇을 돌려줄 것인가

레이트 리밋 발동과 쿼터 초과 발동은, 고객에게 다르게 보여야 한다.

- 레이트 리밋. 429를, 몇 초 동안 기다리라고 말해주는 Retry-After 헤더와 함께. 이건 "느려"라는 신호이고 예상한 것이요, 일시적인 것이다. 고객의 코드가 다뤄야 한다.
- 쿼터 초과. 429가 아니다. 테넌트는 월 허용량을 다 쓴 것이다. 무엇을 다 썼는지, 그리고 무엇을 해야 하는지를 말하는 메시지, "저장공간 100% 사용, Pro로 올리면 20GB"를 갖고 403 혹은 402를 돌려준다. 고객은 1초 뒤에 retry하는 게 아니라, 결정을 내려야 한다. 여기에서 메시지도 없는 429를 돌려주면, 고객의 retry logic이 한 달 내내 당신을 때린다.

그리고 hard stop의 앞에 soft stage가 있다. 어떤 meter든 90%에서, 고객이 통지를 받는다. email이든, in-app 알림이든, 갖고 있는 수단이 무엇이든. 아무도 100%에서 hard stop을 처음에 알게 되면 안 된다.

3.7 크래시를 견디는 metering

meter는 memory에서만 살아서 안 된다. 배포 시 reset되는 counter는 mood일 뿐이다. durable한 pattern은 이렇다.

- 사용 event를 테이블이나 큐에 append한다. (tenant_id, meter, amount, ts)
- event를 스케줄로 daily total로 올린다.
- 월 청구는 daily rollup을 읽는다.
- raw event는 한정한 기간만 보관한다. 예를 들어 30일 [추정]. 그 후엔 버린다.

저장 meter는 특별하다. 각 write마다 byte를 counter하는 것은 느리고 drift한다. 대신 reconcile. 시간당 잡이 테넌트의 실제 size를 계산해서, 그것을 meter로 write한다. write path는 storage를 meter하지 않고 reconciler가 그것이며, drift는 시간마다 스스로 보정된다.

idempotency가 중요하다. 요청이 retry되면, meter는 두 번 count하면 안 된다. request id를 갖고 append-only인 event, 혹은 처음 성공에서만 increment하는 meter가, 숫자를 솔직하게 지킨다. 데이터로 답할 수 없는 billing dispute는, 잃는 고객이다.

3.8 배치 작업도 count

위의 meter는 요청 path를 위한 것이다. 하지만 background job을, export, report, webhook, OCR을 일으키는 테넌트는, 실제 work를, 실제 compute를 쓰고 있다. 그 job은 call이나 batch meter를 소비하고, 테넌트별 concurrent cap을 갖고 있다. cap은 shape limit이다. 동시에 몇 개의 job을 돌릴 수 있느냐. 그게 4장의 첫 half인데, unlimited concurrent job을 가진 테넌트는, 터질 날만 기다리는 noisy neighbor니까.

3.9 테넌트 안의 fairness

마지막 edge 하나. 한 테넌트는 여러 API key를 가질 수 있다. service account마다 하나, integration마다 하나. limit은, per-key로 판매하는 게 아니라면, 테넌트에 있고 key에 있지 않다. 한도 미만의 40개 key를 갖고 합치면 tier의 10배인 고객은, meter의 bug가 아니다. tier보다 많이 쓰는 고객이다. meter는 테넌트에서 합산하고 통지는 테넌트를 가리키고 대화는 계정을 가진 사람과 한다.

3.10 quota check는 어디에 놓나

레이트 리밋은 요청의 앞에 있다. quota check는 한 단계 늦다.

요청이 올 때, 이 요청이 meter를 얼마나 늘리는지 알 수 있어야 한다. "문서 하나 저장"은 저장 meter를, "웹훅 발송"은 호출 meter를 늘린다. 그래서 handler가 일한 뒤에, 늘린 양을 event로 append한다. 요청당 event 한 줄.

그 event가 쌓여서 daily total이 되고, total이 티어 한도와 비교되어, soft stage와 hard stop이 결정된다. 즉, 개별 요청은 "지금 한도를 넘었는가"를 실시간으로 알기 어렵고, daily total이 그 역할을 한다. 저장처럼 reconcile로만 알 수 있는 meter에서는, 하루 단위가 실질적인 체크 주기다.

한 가지 예외가 있다. 좌석. 좌석은 가입 이벤트다. 새로운 team member가 추가될 때, 좌석 meter를 늘리고 티어 한도를 넘으면 추가를 막는다. 좌석은 가입 경로에서 check된다.

meter의 종류에 따라 check의 위치가 달라진다는 것. 그게 실무의 전부다.

3.11 구현 스케치

token bucket을 Redis로 구현하면, 대개 이런 모양이다.

테넌트별 key, 예컨대 rate:{tenant_id}. 버킷의 현재 토큰 수와 마지막 refill 시각을 저장한다. 요청이 오면, refill을 계산하고(지나간 시간 곱하기 refill rate), 토큰이 1 이상이면 하나를 꺼내서 요청을 통과시키고, 0이면 429를 돌려준다.

이 "계산하고 꺼내고 통과시킨다"는 세 동작은 원자적으로 되어야 한다. 두 요청이 동시에 1토큰을 꺼내면, 과잉 허용이 된다. 그래서 대개 Lua 스크립트로 한 번에 실행하거나, Redis의 원자적 명령으로 묶는다.

quota meter는 더 느긋하다. event를 append할 때, 지금 total이 한도인지를 굳이 매번 계산하지 않아도 된다. daily rollup이 그 일을 하니까. 하지만 soft stage(90%) 통지는, append 시점에 대략적으로 확인해서, 그 테넌트가 오늘 통지를 아직 안 받았다면 보낸다. exact하지 않아도, 대개의 burst는 이 지점에서 잡힌다.

중요한 건, 이 구현이 meter의 "shape"을 정한다는 것. token bucket의 capacity와 refill rate를 티어에서 오게 하면(3장의 표), 구현은 티어를 실행하는 기구가 된다.

3.12 Studio로 돌아와서

1장의 에이전시는 Pro에 있다. 20GB, 월 10만 호출, 50 rps. 화요일 밤 9시에 call quota에 닿는다. Starter의 빵집은, 전혀 눈치채지 못했다. 그게 시스템이 working하는 것이다. meter는 테넌트에 있었고 limit이 burst의 모양을 정했고 통지는 맞는 inbox로 갔고 다른 테넌트의 레이턴시는 움직이지 않았다.

4 4장. noisy neighbor: 한 테넌트가 전체를 먹일 때

2장에서는 경계를 데이터베이스 층에서 그었다. 3장에서는 테넌트별 한도를 숫자로 만들었다. 이 장은 한도에도 불구하고 한 테넌트가 공용 자원을 과도하게 차지해 다른 테넌트가 느려지는 상황을 다룬다. 그게 noisy neighbor다.

4.1 왜 1인 플랫폼에서 더 위험한가

noisy neighbor는 규모가 큰 플랫폼의 문제처럼 보인다. 하지만 1인 개발자의 한 대 서버에서 더 위험하다. 완충할 자원이 적고, 발견이 늦기 때문이다.

완충할 자원이 적다. 클라우드의 큰 풀이라면, 한 테넌트가 한 노드를 채워도 다른 노드가 있다. 한 대 서버에서는 그런 선택지가 없다. 빵집과 에이전시가 같은 CPU, 메모리, 디스크 위에서 돈다. 에이전시가 채우면, 빵집은 그대로 느려진다.

발견이 늦다. 모니터링 팀이 없으면, 자원을 많이 쓰는 테넌트를 알아챌 사람은 당신뿐이다. 그리고 당신은 대개 코드에 집중하고 있다. 그래서 1인 플랫폼에서는, noisy neighbor를 "발견해서 대응"하는 구조가 아니라, "설계로 미리 막아"두는 구조가 필요하다. 이 장의 전부다.

4.2 문제가 생기는 곳

여러 테넌트가 공유하는 자원은 대개 네 곳이다.

데이터베이스 연결 풀. 앱이 데이터베이스에 접속할 때 쓰는 연결은 한정되어 있다. 한 테넌트가 연결을 많이 잡고 있으면, 다른 테넌트는 연결을 얻기 위해 기다린다.

CPU와 메모리. 요청을 처리하는 프로세스는 CPU와 메모리를 쓴다. 무거운 작업이 하나 걸리면, 다른 요청의 처리가 느려진다.

디스크 I/O. 읽기나 쓰기가 몰리면, 모든 테넌트의 입출력이 느려진다.

네트워크. 대역폭이 한정되어 있으면, 한 테넌트의 큰 응답이 다른 테넌트의 작은 요청을 기다리게 한다.

이 네 곳은 전부 하나의 풀이다. 한 테넌트가 풀을 독점하면, 나머지는 줄을 선다.

4.3 DB 연결 풀 cap

가장 흔하고 가장 먼저 다뤄야 하는 곳은 DB 연결 풀이다.

테넌트별로 연결 수 상한을 둔다. 전체 풀이 100개라면, 한 테넌트가 최대 20개까지만 쓰게 한다. 그러면 한 테넌트가 50개를 독점해도, 나머지 테넌트는 80개를 쓸 수 있다.

구현은 대개 연결 풀 앞에 테넌트별 counter를 두는 것이다. 테넌트가 연결을 얻을 때 counter를 증가시키고, 반납할 때 감소시킨다. 상한에 닿으면, 그 테넌트의 새 연결은 대기하거나 거부된다.

여기서 거부하는 건 3장의 레이트 리밋과 다르다. 레이트 리밋은 시간당 요청의 양을 막는 것이고, 이건 지금 이 순간 점유 중인 자원의 개수를 막는 것이다.

4.4 쿼리 타임아웃

연결을 갖고 있는 동안, 그 연결이 쿼리를 얼마나 오래 실행할 수 있는지 제한한다.

테넌트별 쿼리 타임아웃을 세션에 넣는다. 일반 요청은 5초, 배치 작업은 60초처럼. 타임아웃이 지나면 데이터베이스가 쿼리를 취소한다.

느린 쿼리 하나가 연결을 오래 잡고, 그 결과 다른 테넌트의 요청을 기다리게 만들기 때문이다. 타임아웃은 그 잡힌 연결을 놓게 한다. 연결 풀 cap과 함께 쓰면, 한 테넌트가 연결을 오래 독점하는 모양을 둘 다 막을 수 있다.

4.5 캐시라는 다섯 번째 통로

2장의 경계 체크리스트에서 캐시를 잊지 말라고 했다. noisy neighbor의 관점에서도 캐시는 위험하다. 메모리 캐시는 공용이다. 한 테넌트가 큰 응답을 많이 캐시하면, 그 테넌트의 엔트리가 캐시를 채우고 다른 테넌트의 엔트리가 밀려난다. evicted된 엔트리는 miss가 되고 miss는 데이터베이스로 돌아간다. 결국 한 테넌트의 캐시 습관이 다른 테넌트의 DB 부하가 되는 구조다.

해결은 테넌트별 캐시 용량 cap이다. 전체 캐시의 일정 비율, 예를 들어 20%까지만 한 테넌트가 쓰게 한다. cap에 닿으면, 그 테넌트는 cache-through로 데이터베이스를 직접 읽는다. 느려지지만 다른 테넌트를 starve하지는 않는다. 캐시 키에 테넌트 접두사가 있는 2장의 규칙이, 여기서 비로소 용량 관리의 전제가 된다.

4.6 background job의 공정한 배분

3장에서 배치 작업도 quota를 쓴다고 했다. 여기서는 동시에 몇 개의 잡을 돌릴 수 있는지, 그리고 어떤 순서로 돌릴 것인지가 문제다.

테넌트별 큐 깊이 cap. 한 테넌트가 천 개의 잡을 넣으면, 그것들이 전부 동시에 돌면 안 된다. 동시에 실행되는 잡의 수를 테넌트별로 제한한다.

가중 우선순위. 모든 테넌트를 똑같이 대우하면, 한 테넌트가 다른 테넌트를 starve할 수 있다. 유료 티어가 높은 테넌트에게 더 높은 우선순위를 주거나, 반대로 공평하게 돌려 한 테넌트가 다른 테넌트를 막지 않게 한다. 어느 쪽을 정할지는 설계 결정이고 그 선택을 계약이나 문서에 쓴다.

4.7 CPU와 메모리

요청을 처리하는 프로세스 층에서는, cgroup 같은 운영체제 수단을 쓸 수 있다. 테넌트별 컨테이너, 혹은 프로세스 그룹에 CPU와 메모리 상한을 두는 식이다.

1인 개발자가 한 서버에서 이걸 완벽하게 하는 건 어렵다. 대개는 "앱을 테넌트별로 분리하지 않는다"는 전제에서, 레이트 리밋과 연결 풀 cap으로 대부분의 문제를 푸는 것이다. cgroup은 정말 무거운 테넌트에만 쓰는 추가 수단이 된다.

4.8 감지: 테넌트별 metric

noisy neighbor를 막으려면, 먼저 누가 소음인지 알아야 한다. 그래서 metric에 테넌트 태그를 달아야 한다.

metric | 뜻 |
요청 수 | 테넌트가 보낸 요청 |
DB 시간 | query가 쓴 시간 |
큐 깊이 | 대기 중인 잡 수 |
CPU 시간 | 프로세스가 쓴 CPU |

이 metric을 보드나 로그에 두고 어느 테넌트가 최근 자원을 많이 썼는지를

살핀다. 3장의 metering과 같은 데이터에서 나온다. meter에 테넌트가 이미 들어가 있으므로, metric도 같은 경로로 태그된다.

4.9 소음을 잡았을 때: runbook

알람이 울리면, 이 순서로 움직인다.

- 확인. metric에서 소음이 어느 테넌트인지 확인한다. 어느 metric이, 어느 기간에, 얼마나 평소보다 큰지를 본다.
- 차단. 그 테넌트의 cap을 임시로 낮춘다. 연결 풀 cap을 줄이고 큐 깊이 cap을 줄이고 필요하면 레이트 리밋을 낮춘다. 다른 테넌트는 여기서 숨을 돌린다.
- 알림. 그 테넌트의 관리자에게, 무슨 일이 있었는지, 왜 그랬는지, 이제 무엇을 어떻게 할지를 쓴다. "당신들이 늦게 해서 서비스 품질이 떨어졌다"가 아니라, "이 metric이 이렇게 올라서, 임시 한도를 이렇게 조정했습니다. 원래대로 돌리려면 이렇게 해주세요."라는 문장이다.
- 해결. 원인이 일시적이면, 한도를 되돌린다. 원인이 구조적이면, 2장의 promotion path로 간다. L3로 올려 물리적으로 분리하거나, 티어 계약을 올려 cap을 높인다.

runbook을 미리 쓰는 이유는, 소음이 터진 밤에 디자인을 재발명하지 않기 위해서다. 소음이 터진 밤에는, 판단이 아니라 절차가 필요하다.

4.10 여유를 두는 설계

cap을 다 달아도, 한 가지 원칙이 남는다. 플랫폼 전체에 여유를 두는 것이다.

서버의 자원을 항상 100% 채워 쓰면, 한 테넌트의 spike가 다른 모든 테넌트의 floor다. 그래서 목표 사용률에 상한을 둔다. 평상시 CPU

사용률이 60%, 데이터베이스 연결 사용률이 70%를 넘지 않도록, 그 전에 스펙업을 한다. [추정: 구체 수치는 워크로드에 따라 달라진다]

이 여유는 비싼 것처럼 보인다. 반대로 말하면, noisy neighbor를 막는 가장 싸고 가장 확실한 수단이다. cap은 한 테넌트를 막지만 여유는 모든 테넌트를 살린다.

4.11 Studio로 돌아와서

1장의 에이전시가 큰 export를 돌려, 디스크 I/O를 채운다고 해보자. 빵집의 페이지가 느려진다. 그때, 에이전시의 export는 background job 큐에 들어가고, 큐 깊이 cap이 적용되어, 동시에 돌리는 export의 수가 제한된다. 그리고 에이전시의 DB 연결이 cap에 닿아, 더 이상의 연결은 대기한다. 캐시도, 에이전시의 엔트리가 cap에 닿아, 나머지 80%는 다른 테넌트의 것이다. 빵집은 몇 초만 늦어질 뿐, 무너지지 않는다. 그게 noisy neighbor 대응이 working하는 모습이다.

5 5장. 테넌트로 돈을 받는 운영: 가격, 과금, 성장

2장에서 경계의 위치를, 3장에서 한도의 숫자를, 4장에서 부하의 한계를 정했다. 이 장은 그 위에 돈을 얹는 일이다. 테넌트 단위의 가격, 청구, 그리고 테넌트가 늘어나면서 달라지는 운영의 모양.

5.1 측정할 수 있는 것만 청구한다

3장의 원칙을 다시 말하면, meter가 없으면 가격이 없다. 저장공간 meter가 없으면 저장공간으로 값을 못 정한다. export meter가 없으면 export를 과금할 수 없다.

이 순서가 많은 1인 개발자를 당황시킨다. 시장은 가격의 출발점이다. "경쟁사는 월 9만 원이네, 우리도 9만 원으로 하자." 그런데 9만 원에 무엇을 포함하는지 정하려면, 결국 측정 가능한 항목을 고르는 작업으로 돌아온다. 그리고 그 항목을 고르는 순간, meter가 필요하다.

그래서 meter-first다. meter를 먼저 정의하고 meter에 한도를 달고(3장), 한도 위에 가격을 둔다. 가격이 meter에 기대어 있으면, 청구서 위의 모든 숫자를 데이터로 답할 수 있다. "왜 이번 달이 더 나왔나요?"라는 질문에, "export가 3,200건이었어요"라고 답하는 시스템이다.

5.2 비용에서 가격으로

티어 가격은 당신의 비용에서 출발해야 한다.

먼저 한 테넌트당 비용이 얼마인지 본다. 서버 비용, 데이터베이스 비용, 백업 저장소, CDN, 이메일 발송. 이 총액을 서비스 중인 테넌트 수로 나눈다.

테넌트 100개에 월 30만 원이면, 한 테넌트당 3천 원이다. [추정: 숫자는 예시일 뿐이다]

그 숫자에 마진을 얹는 것이 가격의 시작이다. 하지만 여기서 멈추면 안 된다. 한 테넌트당 평균 비용은 거짓말이다. 100개 테넌트 중 99개가 100바이트를 쓰고, 1개가 10기가바이트를 쓰면, 평균은 100개 테넌트를 위한 가격이지, 10기가바이트 테넌트를 위한 가격이 아니다. 그래서 meter에 한도를 두고(3장), 한도를 넘는 무거운 테넌트에게 더 높은 가격을 매기는 것이다(4장의 promotion path).

가격의 구조는 대개 이렇다.

- 기본료. meter와 무관하게 매월 내는 돈. 접속, 저장, 좌석 같은 최소한의 것이 포함된다.
- 미터 차액. 기본 한도를 넘는 사용에 대한 돈. export, 추가 좌석, 추가 저장.
- overage. meter가 깨끗한 항목에서, 한도 초과분으로 받는 돈.
- 격리 프리미엄. L3나 L4로 올린 테넌트에게 받는 돈. 2장에서 "영구적으로 더 비싸게 돌리는" 것의 가격이다.

항목을 구분해서 받으면, 각 테넌트가 실제로 쓰는 만큼이 청구서에 드러난다. 그리고 그게 "다음 달에 더 많이 쓰면 더 비싸진다"라는 예측 가능성을 만든다.

5.3 billing loop

청구가 도는 순서를 정하자.

- 일일 집계. 3장의 event가 daily total로 모인다. 테넌트별, meter별, 하루의 숫자.
- 월 청구. 한 달의 daily total을 합쳐, 티어 한도와 비교한다. 한도 안이면 기본료만, 한도를 넘으면 미터 차액이 청구서에 더해진다.

- 중도 변경. 티어를 바꾸면, 변경일 기준으로 prorated한다. 첫 날부터 바뀐 티어의 가격으로, 아니면 남은 일수를 나눠서. 어느 쪽이든, "이번 달은 어떻게 계산했는지"를 한 줄로 설명할 수 있어야 한다.
- 결제가 안 될 때. grace period를 둔다. 예를 들어 7일. 그 사이에는 서비스 유지, 그 다음에는 read-only, 그 다음에는 suspend. 각 단계에서 고객이 받는 메시지가 다르다. "결제가 실패했어요", "read-only로 전환했어요", "계좌가 정지했어요, 데이터는 30일 보관됩니다." [추정: 기간은 정책일 뿐이다]
- 해지. 해지 즉시 서비스 중단. 데이터는 보관 기간만큼, 예컨대 30일, 유지되고 그 후 삭제. 삭제는 2장에서 설계한 "깔끔한 삭제"가 여기선 계약 조항이 된다.

이 loop의 전부다. 그리고 이 loop가 돌아가기 위한 전부가, 3장의 metering이 정직하게 돌아가는 것이다.

5.4 ledger: 청구서의 근거

청구 데이터는 별개의 테이블에서 관리한다. billing ledger. (tenant_id, period, meter, amount, status)의 행.

usage rollup(3장)은 "얼마를 썼는지"이고 ledger는 "얼마를 청구했는지"다. 둘은 다르다. rollup이 100건이고, ledger가 100건이어야 맞는데, 둘이 다르면 어딘가에 버그가 있다. 그래서 월마다 대조하는 reconcile 단계를 둔다. rollup과 ledger가 일치하는지 확인하고, 일치하지 않으면 ledger를 고친다. rollup이 measured truth이니까.

이 테이블 하나를 갖고 있으면, "지난달 청구서가 왜 이래요?"라는 질문을 조회 하나로 답할 수 있다. 그리고 환불이 필요할 때, 환불 대상 행을 찾아가기도 쉽다.

5.5 overage: 멈추거나, 더 받거나

한도를 넘긴 테넌트에 대해 두 선택지가 있다.

멈추기. hard stop. 한도에 닿으면, 더는 쓸 수 없다. 티어를 올리라는 메시지와 함께. 더 받기. overage charge. 한도 넘는 사용량에 대해, unit 가격으로 추가로 청구한다.

1인 개발자에게, 멈추기가 먼저 맞다. overage charge는 정확한 metering, 부분 period의 prorated 모델, "왜 이번 달이 비싸졌죠?"라는 support 대화를 요구한다. 멈추기는 그걸 다 요구하지 않는다. 테넌트가 한도에 닿으면, 3장의 soft stage 통지를 이미 받았고 티어를 올리거나 다음 period를 기다리면 된다.

overage는, meter가 깨끗한 항목에서 나중에 추가한다. call meter처럼, 건수가 정확히 셀 수 있는 것에서. storage처럼, reconcile로만 알 수 있는 것에서는, 멈추기로 시작하는 편이 쉽다.

5.6 격리를 상품으로

2장의 promotion path에 가격이 붙는 순간, 격리는 상품이 된다.

"우리 데이터만 따로 database에 넣어줄 수 있나요?"라는 질문은, 한 테넌트가 당신에게 더 많은 것을 사겠다는 뜻이다. L3는 운영 비용이 더 든다. 마이그레이션이 배가되고 풀 관리가 생긴다(2장). 그래서 L3는 더 비싸게 팔아야 한다. 격리 프리미엄이다.

이게 2장의 seam과 연결된다. L1과 L3가 하나의 repository interface 뒤에서 같이 산다면, "별개 database"는 flag를 바꿈과 가격 행을 더함이다. 같이 살지 못하면, migration project가 되고 project는 margin으로 팔리지 않는다.

5.7 가격 올리는 순간

기존 테넌트에게 가격을 올리는 순간이 온다. 서버가 커졌고 한 테넌트당 비용이 올랐기 때문이다. 이 순간의 원칙은 하나다. 계약된 티어의 가격은, 계약 기간 동안, 바뀌지 않는다.

가격을 올릴 때, 대개 이 중 하나를 고른다.

- 신규만. 새 테넌트부터 새 가격, 기존 테넌트는 유지. 가장 단순하고 가장 안전한 선택이다. 단, "유지"가 영구인지, 다음 갱신까지인지 정해야 한다.
- 갱신 시. 기존 테넌트는 다음 결제 주기부터 새 가격. 30일 전 통지를 대조한다. SaaS의 대다수가 이걸 쓴다.
- 전체 동시. 모든 테넌트에, 같은 날, 새 가격. 통지를 충분히 앞서 보내야 하고 이걸 쓰는 건 대개 비용 급등뿐이다.
- 혼합. 무거운 테넌트만 갱신 시에, 가벼운 테넌트는 한 주기 늦게. 복잡해서, 대개 필요 없다.

1인 개발자에게, 신규만 또는 갱신 시가 대개 맞다. "기존 고객은 그대로, 새 고객은 더 비싸게"는 설명하기 쉬운 이야기이고, "모두가 내일부터 더 비싸다"는 그렇지 않다.

5.8 청구서 예시

Pro 티어의 한 테넌트의 월 청구서를 구체적으로 보자. [추정: 숫자는 예시다]

- 기본료 9만 원. 20GB 저장, 월 100만 call, 팀원 20석까지 포함.
- 저장 meter. 한 달 동안 20GB 안에서 쓰면, 0.
- call meter. 100만 안에 쓰면, 0.
- 좌석 meter. 18명이 쓰면, 0.

- export meter. Pro에 export 2,000건까지 포함인데, 3,400건을 썼으면, 1,400건 곱하기 unit 가격.

합계. 기본료에, export 차액만 더한 청구서. "이번 달이 왜 더 비싸졌나요?"라는 질문에, "export가 1,400건 더 나왔습니다"라고 답한다. meter가 답한다.

이 청구서가 만들어지는 과정의 전부다. 3장의 metering과 이 장의 ledger. meter가 event를 만들고 event가 daily total을 만들고 total이 청구서의 행을 만든다.

5.9 운영 체크리스트

1인 개발자에게 팀이 없다면, 체크리스트가 팀이다.

cross-tenant leak은 P0다. runbook은 이 순서. 발견(2장의 속성 테스트, 혹은 metric 이상), 차단(영향을 받은 경로 disable), 통지(영향을 받은 테넌트에게), 사후 분석. 통지는 선택이 아니다. 늦게 말해주는 테넌트는, 자기 고객에게 먼저 말하게 된다. 그리고 그 버전의 이야기는 당신의 버전보다 나쁘다.

백업은 두 가지. instance 전체 snapshot은 restore를 위해, 테넌트별 logical dump는 deletion을 위해. 둘은 다른 목적을 serve하고 둘 다 필요하다.

관리 경로 접근은 로그된다. 2장의 "별개이고 로그되는" 경로를, 실제로 로그하는 것. 누가, 어느 테넌트를, 언제, 어떤 세션에서. 로그가 없으면, P0의 "발견" 단계가 안 된다.

deletion은 작동한다. export, 보관 기간을 둔 soft delete, hard delete, 그리고 "삭제했습니다"라는 기록. 테넌트가 떠날 때, 답이 날짜여야 한다. project가 아니라.

top talkers review. 4장의 metric을 분기에 한 번 본다. 자원의 30%를

넘는 테넌트가 있으면, 그건 가격 대화의 기회다. [추정: 30%는 임계값 예시다]

5.10 Studio로 돌아와서

빵집은 Starter이고 에이전시는 Pro이고 두 분기째 "우리 데이터 따로"를 요구하던 디자인 스튜디오는, flag를 바꾼 지 며칠 됐다. 세 테넌트가 같은 서버 위에 있다.

빵집은 여전히 스튜디오의 계약서를 못 본다. 에이전시의 느린 밤은 여전히 빵집의 checkout에 닿지 않는다. 그리고 세 테넌트의 청구서는 모두, meter에서 나온 숫자다.

이 책이 가던 곳은 여기였다. 다음 테넌트를 받는 일이 한 줄을 추가하는 일인 시스템. registry에 row가 하나 생기고 meter에 한도가 따라오고 ledger에 숫자가 들어간다. rewrite가 아니라, 한 줄이다. 그리고 빵집은 여전히 스튜디오의 계약서를 못 본다.