설정 규율

환경별 설정과 시크릿을 관리하면서 미치지 않는 기술

ThakiCloud (Hyojung Han)

설정 규율

환경별 설정과 시크릿을 관리하면서 미치지 않는 기술

ThakiCloud (Hyojung Han)

Contents

1 프로덕션이 설정 때문에 무너지는 이유

토요일 밤 11시 반, 알림이 울린다. 결제 성공 메일이 나가지 않는다는 내용.

오후 9시에 평소처럼 배포를 마쳤다. CI는 초록이었고 스테이징에서도 확인했다. 그런데 프로덕션에서만 메일이 멈췄다. 원인은 한 줄. 프로덕션에만 별도로 주입하던 메일 전송용 시크릿의 환경 변수 이름이, 전날 소규모 리팩토링 과정에서 바뀌었던 것. 앱은 새 이름을 읽으려 했지만 프로덕션에는 여전히 구 이름으로만 값이 남아 있다. 서버는 죽지 않았다. 그냥 침묵.

이 이야기는 실제 여러 건의 사고를 섞어 재구성한 것이다. 패턴은 늘 비슷하다. 코드 리뷰에서는 아무도 의심하지 않는 지점, 파이프라인에서는 통과로 표시되는 지점, 그리고 첫 고객이 돈을 내는 지점. 이 곳들이 겹치는 구석에 설정 사고는 숨어 있다.

코드 버그와 설정 사고는 생기는 순간이 다르다. 코드 버그는 리뷰와 테스트를 지나지 못하면 프로덕션에 못 올라간다. 설정 사고는 리뷰 테이블 위에 아예 오르지 않는다. 환경 변수 한 줄은 누군가의 콘솔 작업으로 남는다. 그래서 git 히스토리를 아무리 꼼꼼히 뒤져도 원인이 없다. 원인은 히스토리 밖, 값의 세계에 있다.

1.1 사고의 네 가지 얼굴

설정 때문에 터지는 장애는 대개 네 가지 얼굴을 하고 있다.

첫째, 드리프트다. 환경마다 값이 조금씩 달라지는 상태. 타임아웃이 개발 환경에서는 2초, 스테이징에서는 30초, 프로덕션에서는 300초인 경우를 떠올려 보라. 누가 왜 언제 이렇게 만들었는지는 아무도 모른다. 나중에 값을 올바르게 고치려다 다른 환경의 값을 모른 채 덮어쓰는 일로 이어진다.

차이는 천천히 벌어진다. 그래서 아무도 이상하게 느끼지 못한다.

둘째, 시크릿 누출이다. 데이터베이스 비밀번호가 git 히스토리에 남거나, 에러 로그에 접속 문자열이 찍히거나, 스크린샷에 계정이 그대로 노출되는 경우. 문제는 누출 자체보다, 누출 이후에도 그 값이 계속 유효하다는 점. 값은 죽지 않는다. 당신이 잊기만 하면 된다.

셋째, 환경 혼동이다. 스테이징과 프로덕션이 같은 키를 공유하는 경우. 일단 스테이징에서 돌려보자는 말이 프로덕션 데이터를 건드리는 일이 된다. 경계가 없으니 실수가 없다는 뜻은 아니고, 실수할 문이 늘 열려 있다는 뜻이다.

넷째, 배포 결합이다. 코드와 설정이 항상 함께 움직이는 경우. 값 한 줄을 바꾸려면 배포를 한 번 해야 하고 배포를 하면 의도하지 않은 것도 함께 바뀐다. 그래서 아무도 설정을 가볍게 바꾸지 못하게 된다. 바뀌어야 할 값은 굳어 있고, 굳어져야 할 값은 흔들린다.

얼굴	흔한 트리거	발견 시점
드리프트	값 하나만 고침	장애 발생 후
누출	커밋, 로그, 캡처	외부 보고 때
환경 혼동	키 공유	데이터 이상 시
배포 결합	값 변경 시 배포	롤백 실패 후

네 얼굴의 공통점을 보면, 모두 발견 시점이 늦다. 그 결과가 고객에게 닿은 뒤에야 알게 된다. 설정 사고는 조용히 일어난다. 그리고 조용히 대가를 받는다.

1.2 시스템이 없는 팀의 한 주

추상적으로 말하지 말고 일주일을 따라가 보자.

월요일, 새로운 결제 프로바이더를 연결한다. 개발자 A는 로컬.env 파일에 키를 넣고 같은 파일을 스테이징 배포 스크립트에도 붙인다. 프로덕션 키는 나중에 넣자고 생각한다. 화요일, 프로덕션에 키를 넣으려는데 콘솔에 어떤 이름을 쓰는지가 기억나지 않는다. 스테이징 값과 이름이 조금 달라서 확인하는 데 40분이 걸린다.

수요일, 운영자 B가 타임아웃 에러가 난다고 한다. 프로덕션의 한 값을 300으로 올려야겠다고 판단한다. B는 값을 올리고 그 일이 끝났다고 생각한다. 하지만 그 값이 개발 환경에서는 의도적으로 2초로 되어 있는 이유를 아는 사람은 아무도 없다. 목요일, 개발 환경에서 무언가가 이상해지고, A가 값을 다시 내린다. 누가 언제 왜 바꿨는지 묻는 질문에, 둘 다 모르겠다고 답한다.

금요일, 프로바이더가 키 로테이션을 예고한다. 90일 뒤에 현 키가 무효화된다는 내용이다. 팀은 이 메일을 보고 아무도 일정을 잡지 않는다. 90일 뒤, 시스템이 새벽에 죽는다. 그리고 그 새벽, 드디어 팀은 이 책의 내용을 알기 시작한다.

이 이야기에서 나온 값은 모두 5개 남짓이다. 키 하나, 타임아웃 하나, 로깅 레벨 하나, 리다이렉트 URL 하나, 만료일 하나. 값의 개수가 적을 때조차, 체계가 없으면 매 주 같은 소모가 반복된다. 설정은 많지 않을수록 관리가 쉽다, 라는 직감은 반만 맞다. 값이 적으면 사고는 덜 나지만 값이 많든 적든 체계가 없으면 비용은 매 주 발생한다.

1.3 설정의 진짜 비용

설정의 대가는 장애로만 오지 않는다. 장애는 눈에 보이니까 오히려 고마울 정도다. 더 비싼 비용은 매일의 마찰이다.

가장 흔한 마찰은 확인의 시간이다. 어떤 값이 어디에 있는지 몰라, 5분, 10분, 40분을 쓴다. 한 번에 크지 않다. 하지만 그 확인이 하루 3번씩

한 달을 계속되면, 팀의 생산력을 줄이는 고정 비용이 된다. 값을 20개 정도 가진 작은 서비스가 혼자 운영하는 팀이라면, 주당 30분에서 1시간을 확인에 쓰는 것은 흔한 수준이다 [추정].

두 번째 마찰은 온보딩이다. 새로운 사람이 들어오면, 코드베이스를 읽는 것보다 먼저 필요한 것은 값의 지도다. 이 서비스는 어떤 환경에서 돌고 각 환경에 어떤 값이 주입되고 시크릿은 어디에 있는가. 이 답이 문서화되어 있지 않으면, 온보딩은 구전으로만 진행된다. 그리고 구전은 한 사람이 퇴직하면 끊긴다.

세 번째 마찰은 불안이다. 이 책을 읽고 있는 당신이라면 안다. 환경 값을 하나 고칠 때, 시크릿을 하나 추가할 때 드는 그 느낌. 다른 환경에 영향이 갈까, 누가 이 값을 알고 있을까, 90일 뒤에 뭐가 터질까. 이 불안은 체계 부재의 증세다. 체계가 있으면 불안은 사라지지 않더라도, 확인 가능한 형태로 줄어든다.

비용은 값의 개수만으로는 정해지지 않는다. 환경의 개수와 값의 개수가 곱해진다. 값 10개에 환경 2개면 20개의 조합이다. 값 10개에 환경 5개면 50개. 같은 팀도 환경을 늘리는 순간, 관리해야 할 값의 조합이 눈덩이처럼 커진다. 그래서 1인 개발자의 시스템은 값보다 환경을 먼저 다듬는 것이 늘 이롭다.

1.4 시스템이 있는지 확인하는 법

어떤 팀은 이미 체계가 있다. 다만 이름 붙이지 않았을 뿐이다. 네 가지 신호를 확인해 보라.

첫째, 새 사람이 30분 안에 값을 찾아낼 수 있느냐. 앱이 읽는 환경 변수의 목록이 어디에 있는지, 그 옆에 환경별 값이 있는지, 시크릿은 어디에 있는지. 찾는 데 30분이면 체계가 있는 셈이다.

둘째, 값을 바꾸는 것이 PR이 되느냐. 리뷰를 거치는 변경이 되는 순간,

값에는 히스토리가 생긴다. 히스토리가 있으면 사고가 나도 되돌아갈 길이 있다.

셋째, 시크릿의 만료일을 알 수 있느냐. 데이터베이스 비밀번호의 다음 로테이션 날짜를, 검색 두 번이면 알려면 체계다. 모름이면 장례를 기다리는 셈이다.

넷째, 환경마다 스키마가 같은가. 스테이징에 있는 키가 프로덕션에 없는, 혹은 그 반대인 경우가 하나도 없다면 좋은 출발이다.

네 신호 가운데 하나라도 통과하면, 이 책은 당신에게 훨씬 빨리 와 닿을 것이다. 하나도 통과하지 못했다면, 바로 다음 장부터 천천히 따라오길 바란다.

1.5 12-factor에서 가져온 한 조각

2011년, 헤록쿠(37signals) 팀은 클라우드 앱 작성법을 12개 항목으로 정리한 가이드라인을 공개했다. 그중 설정 관련 항목의 요지는 간단하다.

앱 바이너리는 어디에서든 같아야 한다. 환경이 다른 것은 주입된 값뿐이어야 한다. 값이 환경마다 달라진다면 그 값은 코드 바깥에 두어야 한다. 이 원칙은 더 나아간다. 여기가 프로덕션이라는 판단 자체도 코드에 새겨져서 안 된다. if PRODUCTION: 같은 분기문은 환경 개념을 코드 안으로 끌어들인다. 코드 한 줄을 고쳐도 모든 환경의 행동이 달라질 수 있다면, 그 앱은 완전히 테스트할 수 없다.

구체적으로 보자. 아래 코드는 환경 개념을 코드 안으로 넣은 예다.

if os.Getenv(“ENV”) == “production” { timeout = 300 } else { timeout = 2 }

같은 앱이지만 ENV 값에 따라 바이너리의 행동이 갈라진다. 이 구조에서 ENV를 잘못 주입하면, 타임아웃은 물론이고 코드에 흩어 있는 모든

분기문이 다른 세계로 넘어간다. 반면 값 자체를 주입받게 하면, 이런 분기문은 사라진다.

timeout := envInt(“TIMEOUT”, 300)

바이너리는 어디에서든 같다. 바뀌는 것은 TIMEOUT의 값뿐이다. 이 차이가 12-factor가 말하는 설정의 핵심이다.

이 가이드라인은 출발점이다. 팀 규모와 기술 스택에 따라 일부는 그대로 쓰고, 일부는 바꿔 써야 한다. 이 책에서 12-factor를 인용할 일은 거의 없다. 대신 값을 바깥으로 꺼낸다는 원칙 하나를 받아, 혼자서도 운영할 수 있을 만큼 구체화할 것이다. 원칙이 주는 것은 방향이지, 값 하나하나의 거처를 정하는 절차는 아니다. 그 절차를 세우는 것이 이 책의 일이다.

그렇다면 설정 규율이 정확히 무엇인가. 여기서 정의를 내려 보자. 앱이 소비하는 모든 값에 대해, 그 값이 어디에 있는지, 누가 바꿀 수 있는지, 환경 사이로 어떻게 이동하는지를 정하는 규칙의 집합이다. 핵심은 모든 값이라는 말에 있다. 시크릿에만 적용하고 타임아웃이나 URL에는 적용하지 않으면, 그것은 패치다. 프로덕션에만 적용하고 개발에는 적용하지 않으면, 결국 개발 환경의 값이 프로덕션의 값을 다시 오염시키게 된다.

1.6 이 책의 골격

2장에서는 값을 4개의 층으로 분리하는 법을 다룬다. 코드 안에 두는 값, 환경별로 다른 값, 시크릿, 켜고 끄는 값. 어떤 값이 어느 층에 들어가는지를 4개의 질문으로 판정하는 절차가 핵심이다. 이 장만 제대로 돌아가면, 설정에 대한 팀 내 논쟁의 대부분이 사라진다.

3장에서는 시크릿을 다룬다. 시크릿은 값 중에서도 특히 손이 많이 간다. 왜 자주 새는지, 로테이션을 어떻게 일상이 만들지, 이미 새어 나간 값을 발견했을 때 몇 시간 안에 무엇을 해야 하는지까지. 순서가 중요하다.

조사를 먼저 하다가 새 값을 그대로 둔 팀은 두 번 사고를 낸 팀이다.

4장에서는 환경 분리, 피쳐 플래그, 드리프트 방지를 다룬다. 배포와 릴리스를 나누는 법, 프로덕션 값이 슬쩍 달라지는 것을 알아채는 법, 그리고 매주 10분으로 유지하는 점검 루틴. 작은 반복이 핵심이다.

1.7 오늘 밤에 해 볼 일

책 전체를 이해한 다음에 시작할 필요는 없다. 오늘 밤 10분이면 된다.

앱이 읽는 환경 변수를 전부 목록으로 뽑아라. 코드에서 getenv, os.Getenv 같은 호출을 검색하면 된다. 그 옆에 열 세 개만 채우면 된다. 값이 환경마다 다른가. 민감한 값인가. 최근 한 달 사이 바뀐 적 있나.

세 열 가운데 하나라도 모름으로 채워진 값이 있다면, 그것이 바로 이번 주에 먼저 손대야 할 값이다. 알 수 없다는 것 자체가, 그 값은 아직 아무 체계에도 속하지 않는다는 뜻이다. 그리고 알 수 없는 값이 쌓인 시스템은, 어느 주말 밤에 너를 부를 준비를 마친 시스템이다.

2 각 값은 어디에 둘 것인가

설정이라고 하면 대개 환경 변수를 떠올린다. 그러나 앱이 소비하는 값의 종류는 그보다 넓다. 상수로 쓰이는 숫자, 시크릿, 기능을 켜고 끄는 스위치까지. 이 장에서는 값을 4개의 층으로 나눈다. 층을 나누면, 값 하나가 어디에 있어야 하는지가 절차가 된다.

2.1 4개의 층

1층, 코드 상수. 모든 환경에서 같고 운영 중에는 바뀌지 않는 값. 기본 페이지 크기, 알고리즘 내부 계수, 프로토콜 버전 같은 것들. 이런 값은 코드에 적는 것이 맞다. 바꿀 일이 생긴다면, 상수라는 전제가 무너진 것이다. 그때 옮겨도 늦지 않다. 중요한 것은 상수를 상수로 쓰는 근거다. 앞으로 같을 것이기 때문이다.

2층, 환경 설정. 환경마다 값이 다르지만 새는 것 자체로는 큰 손해가 없는 값. 데이터베이스 호스트, 타임아웃, 로깅 레벨, 외부 API 엔드포인트. .env 파일이나 클라우드의 환경 변수 저장소에 둔다. 2층 값은 git에 들어갈 수 있다. 값은 제외하고. 다음 장에서 보는 시크릿과 2층 값을 구분하는 기준은, 알려지는 것만으로 손해가 나는가 이다.

3층, 시크릿. 새면 접근이나 위조를 바로 가능하게 하는 값. 비밀번호, API 토큰, 서명 키, 클라이언트 시크릿, TLS 사측 키. 시크릿 매니저나 클라우드 키 저장소에 두고 코드와 리포지토리에 절대 두지 않는다. 3층은 이 책 3장의 전제다. 여기서 미리 말해 둘 것 하나: 시크릿은 상태다. 로테이션하면 구 값과 새 값이 한때 공존하기 때문이다.

4층, 피쳐 플래그. 기능의 켜짐과 꺼짐을 결정하는 값. 배포를 거치지 않고 바꾸고 보통 서비스 전체의 행동에 영향을 준다. 4장에서 더 다룬다.

4층 값은 환경 설정과 겹치는 경우가 많은데, 구분 기준은 변경의 성격이다. 환경 설정은 이 환경에서는 이렇게 동작한다는 선언이고, 플래그는 지금 이 기능을 켤 것인가라는 결정이다.

2.2 왜 층을 나누는가

한 가지 파일에 모든 값을 얹어 두는 방식이 있다. 하나의 큰.env, 혹은 하나의 큰 설정 파일. 처음에는 편하다. 값이 열 개 남짓이면, 파일 하나를 열어 보면 다 보인다. 그런데 값이 오십을 넘기고 환경이 두 개를 넘기는 순간, 그 파일은 읽기보다 검색해야 하는 대상이 된다.

층을 나누는 이유는 검색을 없애기 위해서다. 2층 값이 어디에 있는지, 3층 값이 어디에 있는지, 4층 값이 어디에 있는지가 각각 하나의 장소로 묶이면, 질문이 좁아진다. 이 값을 확인해야 한다는 질문은, 이 값이 몇 층인가라는 질문으로 바뀐다. 그리고 층이 정해지면 장소가 정해진다. 장소가 정해지면, 검색이 불필요해진다.

층은 약속이다. 4층이라는 이름이 붙기만 한 구조는 아무 효력이 없다. 효력이 생기는 것은, 팀이 이 값을 2층에 둔다는 말을 자연스럽게 하는 순간이다. 그 순간부터, 2층에 없는 값은 2층이 아니라는 검증을 받아야 한다. 검증이 자연스러워진 체계는, 사고를 줄인다.

2.3 판정 절차: 4개의 질문

새 값을 추가하거나 기존 값을 옮길 때는, 이 네 질문을 순서대로 던진다.

질문 1. 환경마다 값이 다른가. 모든 환경에서 같다면 1층이다. 코드에 적자. 여기서 함정이 있다. 지금은 같다는 말은 영원히 같다는 말이 아니다. 다음 분기에 지역을 하나 더 열면 같지 않게 된다. 바뀌지 않을 확신이 없으면 2층으로 올린다. 1층에 두는 것의 이점은 코드 가독성뿐이므로, 확신이 없으면 과감히 2층으로 넘기는 편이 낫다.

질문 2. 이 값이 새면 손해를 보는가. 네라고 답했다면 3층이다. 시크릿으로 다룬다. 손해가 즉시 발생하지 않아도, 다른 값과 조합되면 문을 여는 열쇠가 될 수 있다. 관리자 계정의 사용자 이름은 새도 아무 일 없는 값처럼 보이지만, 비밀번호와 결합하면 공격의 절반이 끝난 상태가 된다. 손해가 발생하는 메커니즘이 있는지로 판정한다.

질문 3. 배포를 하지 않고도 바꿔야 하는가. 바꿔야 한다면 4층, 피쳐 플래그다. 2층 값도 런타임에 바꿀 수 있는데, 그때는 변경 주기를 본다. 한 달에 한 번 정도면 설정이고 한 주에 여러 번이면 플래그다. 주기가 불분명하면 2층에 두고 관찰한다. 한 달 후에 주기가 분명해지면 그때 층을 옮긴다. 층을 옮기는 것은 값의 이동이지, 새 값의 생성이 아니다.

질문 4. 사람이나 테넌트마다 다른가. 다르면 데이터다. 데이터베이스나 테넌트 설정 테이블에 둔다. 이 질문을 빼먹으면 사고가 난다. 고객사별로 결제 게이트웨이가 다른데, 그 값을 환경 변수로 올리려 해 본 사람이 있을 것이다. 환경 변수는 서비스 하나당 값이 하나다. 고객사마다 다른 값은 그 정의에 맞지 않는다. 데이터를 설정으로 올리면, 고객사가 늘 때마다 환경 변수가 늘고 환경 변수가 늘 때마다 4장의 모든 절차가 무거워진다.

2.4 예: 주문 서비스의 값을 걸러보기

가상의 주문 서비스를 하나 만든다. 결제, 재고 이메일 전송을 담당하는 작은 서비스다. 이 서비스가 읽는 값을 전부 꺼내, 4개의 질문을 통과시켜 본다.

값	층	이유
DB 호스트	2	환경마다 다름
DB 비밀번호	3	새면 즉시 손해
로깅 레벨	2	환경마다 다름
S3 버킷 이름	2	환경마다 다름

값	층	이유
재고 확인 재시도 3회	1	전부 같고 안 변함
페이지 크기 20	1	전부 같고 안 변함
블랙프라이데이 할인	4	배포 없이 켜고 끔
테넌트별 결제사	데이터	테넌트마다 다름

하나씩 이유를 확인하자. DB 호스트는 환경마다 다르지만 주소가 알려졌다고 데이터베이스가 열리는 것은 아니므로 2층. DB 비밀번호는 3층. 로깅 레벨은 개발에서는 debug, 프로덕션에서는 info로 달라지므로 2층. S3 버킷 이름도 환경마다 다르지만 버킷이 공개 읽기만 되지 않는 한, 주입이 잘못되어야 손해가 나므로 2층. 재시도 3회와 페이지 크기 20은 모든 환경에서 같고 고객이 바뀌어도 변할 이유가 없으므로 1층. 페이지 크기 20은 모든 환경에서 같고, 고객이 바뀌어도 변할 이유가 없으므로 1층. 블랙프라이데이 할인은 연 1회, 배포를 거치지 않고 켜고 꺼야 하므로 4층. 테넌트별 결제사는 테넌트마다 달라, 데이터다.

이 목록을 만든 순간, 이 서비스의 값은 6개다. 6개 중 4개가 2층, 1개가 3층, 1개가 4층, 1개가 데이터. 비율은 서비스마다 다르지만 중요한 것은 비율이 아니라, 각 값의 층이 한 문장으로 설명된다는 점이다. 설명이 안 되는 값이 있는 한, 그 서비스의 값은 6개 이상이다.

2.5 흔한 오배치

층을 나눈 다음에는, 오배치를 알아보는 연습을 하자. 오배치는 대개 네 가지 형태로 온다.

첫째, 시크릿을.env에 커밋하는 것. 3층 값을 2층의 자리로 올리는 사고다. 이 장에서는 3층의 정의를 분명히 해 두었고 3장에서는.env.example의 자리에만 이름을 두고, 값을 매니저에 두는 방법을 다룬다.

둘째, 기능의 on/off를 코드 상수로 만드는 것. 4층 값을 1층에 가둔다. 결과로는, 기능을 켜려면 배포를 해야 하고 끄려면 또 배포를 해야 한다. 킬 스위치가 필요한 순간, 배포 파이프라인을 돌리는 팀은 이미 늦은 팀이다.

셋째, 테넌트 값을 환경 변수로 올리는 것. 질문 4를 빼먹은 사고다. 고객사 한 개가 추가될 때마다 환경 변수를 하나 추가하고 프로덕션 콘솔에 손을 댄다. 값의 개수가 고객사의 개수에 비례해서 늘어나는 구조는, 곧 유지 불가능한 구조다.

넷째, 민감하지 않은 값을 시크릿으로 격리하는 것. DB 호스트를 시크릿 매니저에 두는 경우다. 손해를 막는 데는 성공했지만 확인의 비용을 늘렸다. 시크릿 매니저에 접근하려면 별도 절차가 필요하므로, 호스트 하나 확인하는 데 그 절차가 붙는다. 2층 값은 2층에 두는 것이 비용을 줄이는 일이다.

2.6 회색 지대의 값

실제로는 4개의 층에 한 번에 떨어지지 않는 값이 있다. 회색 지대의 값을 세 가지 예로 다룬다.

첫째, 인증이 필요한 외부 API 엔드포인트. URL 자체가 2층 값이고 URL 뒤에 붙는 토큰은 3층 값이다. 둘을 한 값으로 보면 안 된다. 분리해서, ENDPOINT와 TOKEN을 별개의 키로 둔다. ENDPOINT는.env에, TOKEN은 매니저에. 이 분리를 하면, 로테이션은 TOKEN만 건드리면 되고 엔드포인트가 바뀌는 경우는 드물다. 두 값의 주기가 다르므로, 하나로 묶으면 짧은 주기의 값이 긴 주기의 값을 끌어안고 돌아간다.

둘째, 환경별 기본값이 다른 플래그. 블랙프라이데이 할인은 프로덕션에서만 켜도 되지만 개발에서는 항상 켜두면 테스트에 편하다. 이 경우, 플래그의 현재 값은 4층이고 그 기본값은 2층이다. 구조를 이렇게 잡으면, 개발에서는 기본값이 항상 켜져 있고, 프로덕션에서는 명시적으로 켜거나

끄는 값이 있다. 플래그 서비스의 기본값을 환경별로 주입하는 식이다.

셋째, 시크릿으로 격리되었는데도 2층처럼 보이는 값. DB 호스트를 시크릿 매니저에 두는 경우. 이건 오배치 네 가지의 넷째에서 다뤘다. 회색 지대가 아니라, 잘못된 편성이다. 구분하는 기준은, 확인의 주기와 손해의 메커니즘이다.

회색 지대의 값이 보인다면, 그것이 1개인지 2개인지부터 세어 보라. 값이 둘로 쪼개질 수 있는 경우가 대부분이다. 쪼개진 다음에, 각 조각을 질문으로 통과시킨다.

2.7 30초 규칙

이 장의 절차를 한 문장으로 요약하면, 값이 생길 때 4개의 질문을 통과시키지 못한 값은 어디에도 두지 않는다는 것이다.

실전에서는 이렇게 한다. PR 템플릿에 이 값은 몇 층인가, 왜인가 한 줄을 추가한다. 리뷰어가 이 질문에 답이 없는 PR을 통과시키지 않는다. 리뷰어가 매번 묻지 않아도, 2, 3주만 지나면 개발자 스스로가 질문을 던지기 시작한다. 그 순간부터 설정은 절차의 대상이 된다.

또 한 가지, 이 규칙을 유지하는 습관이 있다. 값을 옮길 때, 층이 변했다는 주석을 남기는 것. 1층이 2층으로 옮겨진 값은, 코드 주석에 2026년 8월, 지역 확대 준비로 2층으로 이동이라고 한 줄을 남긴다. 주석이 없으면, 다음에 그 값을 보는 사람이 왜 2층인지 다시 추측해야 한다. 추측은 드리프트의 시작이다.

2.8 인벤토리 템플릿

층을 정한 값은, 인벤토리에 한 줄씩 기록한다. 인벤토리는 표다. 한 서비스당 한 표. 표의 열은 다섯 개다.

항목	예	용도
이름	DB_HOST	식별
층	2	배정 근거
이유	환경마다 다름	30초 검증
소유자	A	변경 시 문의
만료	없음 또는 날짜	로테이션 추적

이 표를 유지하는 비용은 주당 5분도 안 된다. 그러나 이 표가 있는 순간, 4장의 모든 절차가 실행 가능해진다. 새 값이 생기면 한 줄을 추가하고 층이 바뀌면 이유를 고치고 소유자가 바뀌면 이름을 고친다.

표의 가장 중요한 열은 이유다. 층은 예이고 이유는 근거다. 이유 없이 층만 있는 인벤토리는, 다음 사람이 왜 2층인지 다시 추측하게 만든다. 추측은 드리프트의 시작이다.

다음 장에서는 3층, 시크릿에 집중한다. 시크릿은 4개 층 중에서도 취급이 특별하다. 왜 그런지, 그리고 그 특별함이 일상이 어떻게 되는지를 다룬다.

3 시크릿을 다루는 손

3.1 시크릿이란 무엇인가

정의부터 분명하게 한다. 시크릿은, 새면 접근 또는 위조를 가능하게 하는 값이다. 데이터베이스 비밀번호, 서비스 계정 키, OAuth 클라이언트 시크릿, JWT 서명 키, 결제 API 키, TLS 사측 키가 여기에 해당한다.

판단의 기준은 알려지는 것 자체가 문제인가. 타임아웃 300초가 알려지면 아무 일도 일어나지 않는다. DB 비밀번호가 알려지면, 그 순간부터 데이터베이스는 열려 있는 것과 같다. 시크릿이 2층 환경 설정과 구분되는 이유는 여기에 있다. 환경 설정은 잘못 주입되면 장애가 나지만, 알려져도 크게 손해가 없다. 시크릿은 알려지는 것만으로 손실이 발생한다. 그래서 취급이 달라진다.

3.2 시크릿이 새는 길

시크릿은 대개 다섯 길을 통해 새어 나간다.

첫째, git 커밋이다..env 파일을 그대로 올리거나, 접속 문자열을 코드 안에 적어 두고 잊는 경우. 히스토리에 남으면, 값을 지워도 지워진 것이 아니다. 히스토리를 고치는 순간부터, 그 값은 새어 나온 것으로 취급해야 한다.

둘째, 로그다. DEBUG 모드에서 환경 변수를 통째로 찍는 코드, 쿼리 문자열에 포함된 토큰이 에러 메시지로 올라오는 경우. 로그는 시크릿의 가장 흔한 은신처다. 로그를 보는 사람이 많을수록, 시크릿을 보는 사람도 많아진다.

셋째, 캡처다. 콘솔의 스크린샷을 PR이나 채팅에 붙이는 경우. 캡처는 값을 복제하지도, 이동하지도 않는다. 그냥 하나 더 만든다. 원본을 지워도

캡처는 남는다.

넷째, 문서다. 런북이나 설계 문서에 예로 쓴 값이, 실제 값으로 남아 있는 경우. 예를 들어 이라고 쓰면서도, 그 바로 뒤에 실제 키를 붙인 문서가 있다. 문서는 오래 산다. 코드가 바뀌어도, 문서는 한 해 전에 쓰인 값으로 남아 있다.

다섯째, 오래된 코드다. 환경 변수로 옮기기 전에 하드코딩했던 값이, 더 이상 쓰이지 않는 파일에 남아 있는 경우. 죽은 코드도 값을 가진다. 그리고 검색을 하면, 죽은 코드까지 다 나온다.

여기에 공통되는 일이 하나 있다. 새어 나온 값은, 그 값이 어디서 쓰이는지를 알 수 없게 만든다. 프로덕션과 개발이 같은 키를 쓰면, 어느 곳에서 새어 나왔는지를 구분할 수 없다. 그래서 환경 분리는 시크릿 관리의 전제다.

3.3 스스로 해 볼 수 있는 점검

원칙은 알고 있지만 내 시크릿이 실제로 안전한지는 별개의 질문이다. 10분 점검으로 답을 구해 보자. 점검은 검색 두 번과 확인 여덟 번이다.

검색 두 번. 첫째, 리포지토리 전체에서 시크릿 값을 검색한다. git log -S로 히스토리까지. 값이 한 번이라도 히스토리에 남으면, 그 값은 이미 새어 나온 값이다. 둘째, 로그 서비스에서 같은 값을 검색한다. 에러 메시지에 섞여 올라온 경우가 있다.

확인 여덟 번..env가.gitignore에 있는가..env.example에 실제 값이 들어 있지 않은가. CI/CD의 변수 목록에 오래된 값이 남아 있지 않은가. 컨테이너 이미지 레이어에 값이 찍히지 않는가. 문서와 런북에 예로 쓴 값이 가상의 값인가. 채팅 기록에 캡처가 남아 있지 않은가. 전 직원이 쓰던 계정의 키는 무효화되었는가. 시크릿의 보관 위치가 한 곳인가, 둘 이상인가.

검색 두 번 중 하나라도 값이 나온다면, 그 날은 점검이 아니라 대응이다. 1단계, 무효화부터.

3.4 파이프라인에도 시크릿이 있다

시크릿은 앱에만 있는 것이 아니다. 파이프라인에도 있다. 빌드 시 주입되는 값, 테스트용 데이터베이스의 값, 배포에 쓰는 자격 증명. 파이프라인의 시크릿은 앱의 시크릿과 다른 점이 하나 있다. 파이프라인의 시크릿은, 파이프라인의 히스토리에 남는다. 빌드 로그, 실행 기록, 아티팩트.

빌드 로그에 시크릿이 찍히는 경우를 보자. 스크립트가 echo로 환경 변수를 출력하는 경우, 빌드 로그에 값이 남는다. 빌드 로그는 파이프라인의 히스토리다. 앱의 로그보다 오래 산다.

컨테이너 이미지도 마찬가지다. 이미지를 빌드할 때 시크릿을 레이어에 넣으면, 그 레이어는 이미지의 히스토리가 된다. 이미지를 배포한 모든 곳에, 시크릿이 함께 간다. 이미지를 다시 배포한다고 사라지지 않는다. 이미지는 값을 주입받는 것은 배포 시점이어야 하고, 빌드 시점이 아니다. 빌드 시점에 넣는 것은, 값을 이미지에 새는 것과 같다.

3.5 기본 원칙 다섯 가지

원칙 1. git에는 시크릿이 들어가지 않는다..env 파일은 로컬에만 두고,.gitignore에 등록한다. 리포지토리에는.env.example만 둔다. 자리만 있고 값은 없는 파일이다. DATABASE_URL=postgres://user:password@host:54 같은 형태. 예의 값은 가상의 값이어야 한다. 실제 값으로 예시를 만들지 마라.

원칙 2. 한 서비스는 자신의 시크릿만 가진다. 전체 시스템에 쓸 수 있는 만능 키를 만들지 않는다. 주문 서비스는 주문 데이터베이스에만 쓸 수 있는

비밀번호를 가진다. 만능 키는 로테이션의 악몽이다. 하나를 바꾸면 전부 멈춘다. 그리고 만능 키가 새면, 사고의 범위가 시스템 전체가 된다.

원칙 3. 로테이션은 일상이어야 한다. 몇 주에 한 번인가로 묻는 질문이 되어야 한다. 로테이션이 쉬워야 자주 한다. 시크릿을 매니저에 두고 있다면, 로테이션은 값 한 번 바꾸는 일이다. 매니저가 없다면, 로테이션은 배포를 수반하는 일이 된다. 그래서 매니저는 로테이션의 전제다.

원칙 4. 로그에는 이름을 남기고 값을 남기지 않는다. 찍어야 한다면 값의 존재 여부만. DATABASE_URL은 설정됨, 이렇게. 시크릿의 값을 로그에 남기지 않는 것은, 원칙이 아니라 위생이다. 위생은 매일 하는 것이지만 매일 해야 하는 이유가 매일 발생하기 때문이다.

원칙 5. 누출은 가능성으로 두지 않는다. 아마 안 새었을 것이다가 아니라, 이미 새었다고 가정하고 대응한다. 공개 저장소에 올렸던 시크릿은, 스캐너가 몇 분 안에 찾아낸다. 찾아내는 데 몇 분이 걸린다면, 쓰는 데도 몇 분이 걸린다고 보면 된다.

3.6 환경마다 시크릿은 다르다

개발용 키와 프로덕션용 키를 분리한다. 팀이 작을수록, 아까워서 같은 키를 쓰려는 유혹이 있다. 그 유혹을 거절하는 이유는 세 가지다.

첫째, 로테이션이 멈춘다. 키를 공유하면, 키 한 번 바꿀 때 모든 환경에 같이 반영해야 한다. 한 환경이라도 늦으면, 그 사이에 구 키로 접속이 가능하다. 늦은 환경이 개발이든 프로덕션이든, 구 키는 유효하다.

둘째, 사고 원인을 알 수 없다. 프로덕션 로그에서 의심스러운 접속이 보이면, 개발 환경에서 나온 것인지를 가를 수 있어야 한다. 같은 키면 그게 안 된다.

셋째, 누출 범위가 불분명해진다. 개발 머신에 새었다는 것을 알아도,

프로덕션에까지 영향이 있었는지 알 수 없다. 키가 다르면, 범위는 키로 자를 수 있다.

3.7 어떤 도구를 쓸 것인가

상황	도구 예	이유
시작, 1인 운영	CI/CD 시크릿	별도 프로세스 없음
여러 서비스	시크릿 매니저	접근 로그와 만료
암호화 키 중심	외부 KMS	키가 밖으로 안 나옴

시작 단계라면, CI/CD 플랫폼 자체의 시크릿 저장소가 충분하다. 별도 서버를 세울 이유가 없다. 서비스가 세 개를 넘어가거나, 시크릿에 접근한 사람이 언제 무엇을 봤는지 기록이 필요해지면, 시크릿 매니저를 도입한다. Vault 같은 것. 여기서 중요한 것은 브랜드가 아니라 두 기능이다. 접근 감사 로그, 그리고 만료와 로테이션 지원.

외부 KMS는, 키가 시스템 안에 들어와서는 안 되는 경우다. 결제 토큰의 서명 키나, 사용자 데이터의 암호화 키. KMS를 쓰면, 키는 KMS 안에만 있고 시스템은 키의 해시나 서명 결과만 가진다. 키를 아는 것이 아니라, 키를 쓰는 구조다.

3.8 새어 나간 값을 발견했을 때

발견은 대부분 우연이다. 공개 스캐너의 알림, 보안 점검, 혹은 전 직원의 제보. 그때 순서가 중요하다.

1단계, 먼저 무효화한다. 새어 나간 값 자체를 로테이션한다. 조사 전에. 조사는 몇 시간 걸린다. 그 동안 구 값은 여전히 열쇠다.

2단계, 범위를 확인한다. git 히스토리, 로그 서비스, 에러 트래커, 그리고 그 값이 공유된 곳. 채팅, 스크린샷, 문서. 언제, 누구의 눈에 닿았는지 시간순으로 적는다.

3단계, 연결된 값을 확인한다. 그 키를 다른 서비스나 다른 계정에 쓰던 것이 있다면, 그것도 함께 무효화한다. 키가 한 곳에 있으면, 사고는 한 곳에서 끝난다.

4단계, 기록한다. 시간이 줄, 무엇이 새었는지, 무엇이 노출되었는지, 무엇을 했는지. 피해가 없었더라도 쓴다. 기록이 있어야 다음에 범위를 빠르게 가를 수 있다.

5단계, 길을 막는다. 값은 어느 길을 통해 나온가. 대부분은 다섯 길 가운데 하나다. 그 곳에 가드를 둔다. 예를 들어, 커밋 전에 시크릿을 검색하는 훅, 로그에서 특정 패턴을 마스킹하는 필터, 문서 템플릿에서 예 값을 가상의 값으로 바꾸는 규칙.

3.9 시크릿도 드리프트한다

시크릿은 값이라기보다 상태에 가깝다. 로테이션하면 구 값과 새 값이 한때 공존한다. 절반의 서비스가 새 값을, 나머지 절반이 구 값을 쓰고 있다. 그 상태로 한 달을 보내면, 어느 것이 유효한지 아는 사람은 없다.

방책은 인벤토리다. 시크릿 하나당 한 줄. 이름, 소유 서비스, 만료일, 마지막 로테이션 날짜, 보관 위치. 10줄이면 10분이고 30줄이면 30분이다. 매주 한 번, 로테이션 기한이 가까운 것을 확인한다.

유형	로테이션 기준	포인트
DB 비밀번호	90일	계정 분리
API 토큰	30일	최소 권한
서명 키	1년	이중 키 전환

유형	로테이션 기준	포인트
클라이언트 시크릿	6개월	환경 분리

로테이션 기준은, 새 값이 유효해지는 동안 구 값도 유효한 기간, 즉 이중 유효 기간과 함께 잡아야 한다. 90일마다 바꾼다면, 90일 동안 구 값도 유효하다. 그 기간 동안 구 값이 쓰이는 곳이 어디인지, 인벤토리가 말해 줄 것이다.

3.10 분기에 한 번, 유출 훈련

대응 스크립트를 한 번도 안 써 본 팀은, 쓸 때 처음 쓰는 팀이다. 처음 쓰는 순간은 보통 주말 밤이다. 그래서 분기에 한 번, 훈련을 하자.

방식은 간단하다. 시크릿 인벤토리에서 값을 하나 고른다. 그 값을 이미 새어 나왔다고 가정한다. 1단계부터 5단계까지, 실제 순서대로 실행한다. 무효화, 범위 확인, 연결 값 확인, 기록, 길 막기. 시간을 재고 막힌 지점을 적는다.

훈련에서 막히는 지점은 대개 두 곳이다. 첫 번째, 2단계의 범위 확인이 느리다. 로그를 어디서부터 봐야 하는지, 어떤 검색어를 쓰는지 모르기 때문이다. 두 번째, 5단계의 길 막기가 없다. 가드를 둘 자리가 정해져 있지 않기 때문이다.

훈련은 다음 분기의 대응 시간을 줄인다. 그리고 대응 시간을 줄인다는 것은, 주말 밤의 너를 지키는 일이다.

3.11 오늘부터

시크릿을 하나 추가할 때, 이 세 동작을 함께 한다..env.example에 이름을 쓴다. 시크릿 매니저 또는 CI/CD 저장소에 값을 둔다. 인벤토리에 한 줄을

추가한다. 세 동작이 한 세트다. 값을 두고 세 동작을 건너뛰는 순간, 그 값은 다음 누출 사고의 후보가 된다.

레벨을 한 단계 올리자면, 기존 시크릿에 대해서도 이 세 동작을 역으로 적용해 보라..env.example에 이름이 없는 시크릿, 인벤토리에 없는 시크릿, 만료일이 없는 시크릿. 세 가지가 하나라도 있으면, 그 시크릿은 오늘 추가한 것이 아니라, 어제부터 방치한 것이다.

4 환경 분리, 피쳐 플래그, 드리프트 방지

4.1 환경이란 값의 묶음이다

환경을 서버 묶음으로 생각하면 실수가 나온다. 환경은 값의 묶음. 설정 값, 시크릿, 인프라 식별자가 함께 묶인 것이다. 이 묶음의 정의가 모호하면, 모든 사고의 원료가 모호해진다. 어느 환경에서 돌았는지, 어떤 값으로 돌았는지, 그 값을 누가 언제 바꿨는지를 묻는 세 질문에 답할 수 없게 된다.

표준적인 세 환경을 정해 둔다.

환경	역할	포인트
dev	개발과 피드백	빠름이 최우선
staging	배포 전 검증	prod와 같은 스키마
prod	실사용	파이프라인으로만 변경

여기에 필요하면 preview를 더할 수 있다. 브랜치마다 일시적으로 만드는 환경. 브랜치의 설정을 prod와 비교하기 전에, 그 브랜치가 만든 값으로 앱을 돌려보는 것이다. preview는 값의 묶음을 검증하는 절차다.

환경의 개수가 중요하다. 각 환경의 값 묶음이 분명해야 한다. dev에는 어떤 값이 주입되고 staging에는 어떤 값이 주입되고 prod에는 어떤 값이 주입되는지, 한 장의 표로 설명할 수 있어야 한다.

4.2 환경 묶음의 실제 모습

추상적인 설명을 한 번, 실제 값으로 옮겨 보자. 3장에서 다룬 주문 서비스의 값 묶음을 표로 쓴다. 스키마는 세 환경 모두 같고 값만 다르다.

값	dev	prod
DB_HOST	localhost	prod.db
LOG_LEVEL	debug	info
TIMEOUT	2	300
할인 플래그 기본값	켬	끔

staging은 dev와 prod 사이 어디에 있어도 된다. 다만, 키 목록은 이 표와 같아야 한다. 값은 staging이 prod를 모방하는 것이 원칙이다. staging에서 prod에 없는 값을 쓰면, 그 값은 prod에 올라갈 때 새로 만들어야 한다. 새로 만드는 값은, 검증되지 않은 값이다.

이 표가 환경의 진짜 모습이다. 서버가 아니라, 값의 묶음. 표를 한 장으로 쓸 수 있다면, 환경은 정의된 것이다. 표를 쓰지 못한다면, 환경은 아직 정의되지 않은 것이다.

4.3 핵심 규칙: 스키마를 공유한다

staging과 prod는 스키마를 공유하고 값을 나눈다. 앱이 읽는 키 목록은 한 번 정의하고, 모든 환경이 그 목록을 채운다. 채운 것이 없는 키는 시작할 때 실패해야 한다. 이것이 fail fast다.

required := []string{“DB_HOST”, “DB_PASS”, “LOG_LEVEL”} for _, k := range required { if os.Getenv(k) == “” { log.Fatal(“missing required env var:” + k) } }

앱이 시작할 때, 필요한 키가 하나라도 없으면 죽는다. 죽는 것이 좋다. prod에서 키가 없으면, 앱은 죽는 대신 엉뚱한 값을 쓰며 계속 돈은 번다. 설정 누락이 동작으로 둔갑하는 것보다, 설정 누락이 시작 실패로 드러나는 것이 낫다.

반대 패턴은 기본 값 구조다. 키가 없으면 코드에 적어 둔 기본 값으로 조용히 넘어가는 것.

timeout := envInt(“TIMEOUT”, 300)

개발에서는 편하다. 그러나 이 한 줄은, TIMEOUT을 주입하지 않은 모든 환경을 300으로 만든다. dev의 2초도 300이 되고 staging의 30초도 300이 된다. 기본 값은 환경의 차이를 지운다. 그리고 지워진 차이는, 나중에 장애의 형태로 돌아온다.

기본 값을 쓰고 싶다면, 명시적으로 쓰라. 환경마다 기본 값을 달리 주입하는 구조가 아니라, 키의 존재를 검증하고 없으면 죽는 구조를 먼저 세운다. 기본 값은 그 다음에, 검증된 키의 보조로 넣는 것이다.

4.4 설정도 코드처럼

환경 값의 템플릿은 리포지토리에 둔다. 시크릿은 제외하고. 각 환경의 값이 담긴 파일을 디렉토리로 나눈다. config/staging.env, config/prod.env. 시크릿은 매니저에 있고 파이프라인이 배포 시점에 주입한다.

이 구조의 효과는, 설정 변경이 코드 변경과 같은 길을 걷게 된다는 점이다. 값이 바뀌면 PR이 열린다. 리뷰가 붙는다. staging에서 확인된 후, main에 머지되는 순간 prod에 반영된다.

그리고 문을 닫아야 한다. prod 콘솔에 직접 로그인해서 값을 바꾸는 길이다. 100% 금지가 어렵다면, 예외를 기록하는 규칙을 만든다. 언제, 누구, 왜, 어떤 값. 당일 기록하고 다음 주 파이프라인에 같은 변경을 머지한다. 예외가 쌓이면, 그것이 곧 드리프트의 목록이 된다.

이 구조에서 빠지기 쉬운 지점이 하나 있다. config/staging.env와 config/prod.env가 서로 다르다는 것이다. 두 파일은 스키마를 공유하지만 값은 다르다. 키가 같아야 하는 것은, 키가 아니라 값이다. 키 목록이 같다는

것을 검증하는 방법은, 두 파일을 키만 추출해서 비교하는 것이다.

```
diff <(cut -d= -f1 config/staging.env | sort)
<(cut -d= -f1 config/prod.env | sort)
```

출력이 비면, 스키마는 같다. 출력이 있으면, 그 차이 하나가 다음 환경 사고의 씨앗이다.

4.5 하나의 값을 올려 보내는 과정

프로덕션의 TIMEOUT을 300에서 600으로 올린다고 해 보자. 30초에 끝나는 일이, 이 구조에서는 이렇게 된다.

1. config/prod.env에서 TIMEOUT=300을 TIMEOUT=600으로 바꾼다.
2. PR을 연다. 리뷰어가 한 줄을 본다. 600의 근거를 묻는다.
3. main에 머지한다. 파이프라인이 prod에 600을 주입한다.
4. 배포가 끝난다. diff가 지난 스냅샷과 비교한다. TIMEOUT이 300에서 600으로, 이 한 줄만.
5. 인벤토리의 스냅샷이 업데이트된다.

5단계. 30초가 아니다. 하지만 30초로 끝나는 일에는, 리뷰도, 근거도, diff도, 기록도 없다. 이 구조가 사는 것은, 다음에 TIMEOUT을 다시 올릴 때, 지난 기록을 보는 데 30초가 걸리지 않기 때문이다.

반대 경우를 보자. 3시에 prod 콘솔에 들어가서 TIMEOUT을 600으로 올린 경우. 리뷰는 없고 근거는 메모리에 있고 diff는 없다. 다음 주, 3시 올림을 누가 했는지 묻는 질문에, 대답은 없다. 이것은 드리프트가 아니라, 드리프트의 시작이다.

4.6 피쳐 플래그

플래그의 목적은 하나다. 배포(deploy)와 릴리스(release)를 나누는 것. 코드에는 기능을 포함해 prod까지 올린다. 다만 스위치는 꺼진 상태. 준비가 되면 스위치를 켠다.

패턴은 세 종류다.

릴리스 플래그. 새 기능의 기본값은 꺼짐. 내부, 일부 고객, 전체. 이렇게 켠다. 켤 때마다 지표 하나를 본다. 에러율, 응답시간, 전환율 중 하나. 지표를 안 보고 켜는 것은, 릴리스가 아니라 배포다.

킬 스위치. 이미 켜져 있는 기능을 사고가 나면 즉시 끄는 값. 배포를 롤백하는 것보다 빨리, 가 존재 이유다. 롤백은 코드 전체를 되돌린다. 킬 스위치는 기능 하나만 끈다.

실험 플래그. 트래픽의 백분율로 켜는 값. 5%, 20%, 100%. 지표를 보고 백분율을 올린다. 실험이 끝나면, 플래그를 지운다. 지우지 않은 실험 플래그는, 코드 속의 분기문이 되어 영주한다.

플래그의 규칙도 세 가지다.

첫째, 모든 플래그는 주인과 기한을 가진다. 기한이 없는 플래그는 코드 속의 분기문이 되어 영주한다. 플래그를 지우지 않으면, 플래그가 당신을 관리하기 시작한다.

둘째, 예산을 정한다. 한 서비스의 활성 플래그는 20개 안쪽이 적당하다 [추정]. 넘으면 기능을 통합하거나 삭제해야 한다는 신호다.

셋째, 플래그는 값이 아니다. prod에서만 켠은, 플래그가 아니라 2층 설정의 문제다. 고객사 A만 켠은, 데이터의 문제다. 플래그는 행동의 on/off에만 쓴다.

플래그의 수명은 이렇게 관리한다.

단계	동작	기준
생성	주인과 기한 기록	기한 없을 시 PR 불가
확장	백분율 올림	지표 확인
완료	전체 켜고	지표 안정
청소	플래그 삭제	기한 도래 시

청소 단계가 빠지는 경우가 많다. 전체로 켠 기능을, 왜 플래그로 남겨 두는가. 전체로 켠 순간, 플래그는 더 이상 결정을 하지 않는다. 결정하지 않는 분기문은, 읽기 어려운 코드다.

4.7 드리프트를 잡는 네 가지 기구

드리프트는 의도해서 생기는 것이 아니다. 값을 조금 바꾸고 잊고 다른 환경도 모르게 따라 바꾸지 않고, 그렇게 차츰 벌어지는 것이다. 잡는 기구는 네 개인다.

인벤토리. 배포할 때, 실제로 적용된 값의 스냅샷을 남긴다. 시크릿은 마스킹해서. 이번 배포와 지난 배포의 값을 비교할 수 있게 된다.

diff. 스냅샷끼리 차이를 본다. 의도한 PR이 아니라면, 그 차이가 바로 드리프트다. 차이 보고를 배포 기록에 첨부한다.

변경 단일 경로. prod 값이 바뀔 수 있는 길은 하나, 파이프라인. 콘솔은 닫는다.

주 1회 조율. 인벤토리, 플래그, 예외 기록을 한 번 본다. 기한이 지난 플래그, 로테이션이 임박한 시크릿, 기록되지 않은 prod 변경.

기구	주기	결과물
인벤토리	각 배포	마스킹 스냅샷

기구	주기	결과물
diff	각 배포	변경 보고서
단일 경로	상시	콘솔 접근 기록
조율	주 1회	예외 목록

네 기구 중 가장 싼 것은 diff다. 스냅샷이 이미 있고 비교는 스크립트 한 줄이므로. diff가 없으면, 인벤토리는 쌓이기만 하고 쌓인 스냅샷은 아무도 보지 않는다. 보는 것이 없으면, 보는 것과 안 보는 것은 같다.

4.8 드리프트의 경계선

드리프트는 의도하지 않은 변경이다. 의도했지만 기록되지 않은 변경은, 아직 드리프트가 아니다. 그러나 드리프트의 후보다. 후보와 현행범을 가르는 경계선은 하나다. 다음 배포의 diff에, 그 변경이 보이는가.

diff에 의도한 PR의 변경만 보이면, 체계는 살아 있다. diff에 PR에 없는 변경이 보이면, 그 변경은 어디에서 왔는가. 대개는 세 곳이다. 콘솔의 직접 편집, 로테이션 후에 안 넘어온 값, 그리고 누군가의 임시 조치.

임시 조치는 특히 위험하다. 사고를 막기 위해 넣은 값이, 사고가 끝난 뒤에 남는 경우. 임시로 넣은 값은, 임시로 넣은 이유와 함께 기록되어야 한다. 이유를 기록하면, 다음 조율에서 지울 수 있다. 이유를 기록하지 않으면, 그 값은 영구로 남는다. 그리고 영구로 남은 값이, 다음 사고의 시작이 된다.

경계선을 지키는 비용은, diff를 보는 1분이다. 1분을 안 보면, diff는 쌓인다. 쌓인 diff는, 읽지 않는다. 읽지 않은 diff는, 없는 것과 같다.

4.9 10분 주간 루틴

매주 같은 요일에 10분을 쓴다.

1. 시크릿 인벤토리에서 로테이션 기한이 한 달 안쪽인 것을 확인한다.
2. 활성 플래그의 주인과 기한을 확인한다.
3. prod 변경 감사에서 파이프라인 밖의 변경이 없었는지 확인한다.
4. staging과 prod의 스키마 검증이 통과했는지 확인한다.

네 항목 중 하나라도 확인 불가가 나오면, 그 주 30분을 더 쓴다. 확인 불가가 쌓이는 것이 드리프트다. 확인 가능한 상태를 유지하는 것이, 드리프트를 막는 상태다.

4.10 마무리

설정 규율은 재능이 아니라 반복이다. 2장의 질문으로 값을 분류하고 3장의 세 동작으로 시크릿을 추가하고, 4장의 10분으로 주를 닫는다. 이 순서가 세 번만 돌아도, 팀에서는 값이 어디 있어?라는 질문의 대답이 빨라진다.

다음 시크릿을 추가할 때, 오늘 읽은 세 동작부터 해 보자..env.example에 이름, 매니저에 값, 인벤토리에 한 줄. 그리고 다음 주 금요일, 10분을 쓴다. 세 동작과 10분. 이것이 이 책의 전부다.