웹후크의 규율

외부 시스템의 소식을 받고, 믿고, 재전송하는 1인 개발자의 기술

ThakiCloud (Hyojung Han)

웹후크의 규율

외부 시스템의 소식을 받고, 믿고, 재전송하는 1인 개발자의 기술

ThakiCloud (Hyojung Han)

Contents

1 통신문을 여는 첫날: 웹후크의 구조와 마음가짐

1인 개발자. SaaS를 만든다. 고객이 Stripe로 결제했다. 결제가 완료된 순간, 제품 이용을 허용해야 한다. 무난한 방법은 1분마다 Stripe API로 "결제가 성공했냐"고 물어보는 것이다. 이게 폴링이다. 소규모라면 돌아가지만 네 가지 문제가 있다.

첫째, 지연이다. 결제가 10초 전에 끝나도, 알아채는 데는 1분까지 걸린다. 고객이 "결제했는데 안 된다"며 메시지를 보낼 수 있다. 둘째, 낭비다. 폴링 횟수는 고객 수에 비례해서 늘고, 응답의 대다수는 "아무일 없었다"다. 고객 수가 세 배가 되면, 쓸모없는 요청도 세 배가 된다. 셋째, 간격보다 빠른 이벤트는 놓친다. 두 번의 폴링 사이에 일어난 일은 나중에 다른 경로로 알아내야 한다. 넷째, 상태 확인 API 자체가 없는 서비스도 있다. 그런 서비스는 일이 일어날 때만 알려주는 식이다.

그러면 보내는 쪽이 연락해 주면 된다. 웹후크가 하는 일이 그것이다. Stripe에 URL을 주고 "이벤트가 생기면 이 주소로 POST 하라"고 한다. 결제가 완료되면 Stripe 서버가 내 엔드포인트로 HTTP 요청을 보낸다. 폴링은 내가 그들의 문을 계속 두드리는 것이라면, 웹후크는 그들이 내 문을 두드리는 것이다.

통제의 방향이 바뀐다. 이건 가장 먼저 몸에 넣어둘 점이다. 폴링에서는 내가 타이밍을 잡고 상대는 답만 한다. 웹후크에서는 상대가 타이밍을 잡고 나는 받아야 한다. 그래서 설계 부담이 "언제 물을지"에서 "아무거나 올 수 있는 것을 어떻게 받을지"로 이동한다.

1.1 폴링과 웹후크, 한눈에

구분	폴링	웹후크
타이밍의 주인	받는 쪽	보내는 쪽
빈 이벤트 비용	매번 지불	없음
첫 인지 시간	간격에 의존	대개 즉시
놓침	간격 사이 발생	없음, 대신 재전송

폴링이 나쁜 건 아니다. 이벤트가 드문데 지연이 신경 쓰이지 않는다면, 5분 간격 폴링이 훨씬 싸고 단순할 수 있다. 웹후크를 세우는 판단 기준은 "현대적이니까"가 아니라, 늦으면 안 되는 일이 있는지, 빈 응답의 비용이 쌓이는지다. 둘 다 해당되면, 편지 상자를 여는 날이다.

1.2 편지 모양

그 "두드림"은 어떤 모양인가. 평범한 HTTP POST 요청이다. 다만 메시지 내용과 몇 개의 헤더가 특이할 뿐이다.

바디는 대개 JSON이다. 이벤트 종류, 그 이벤트를 가리키는 유일한 식별자, 발생한 시각, 그리고 해당 이벤트와 관련된 데이터가 들어 있다. Stripe의 "payment intent succeeded" 이벤트라면, payment intent ID와 금액이 실려 있다.

GitHub도 비슷하다. 저장소에 push happened, pull request가 열렸다, release가 발행됐다. 각각이 이벤트가 되어 내 엔드포인트로 온다. 헤더 X-GitHub-Event가 어떤 이벤트인지 알려 주고, 바디에는 저장소와 행위자 정보가 있다.

중요한 건, 이 "편지"에 고유 ID가 붙어 있다는 점이다. GitHub는 delivery id, Stripe는 evt_로 시작하는 이벤트 ID를 부여한다. 이 ID가 나중에 중복을 다스리고 재전송할 때의 열쇠가 된다. 여기서 기억해 두자.

편지에는 시각도 붙어 있다. 이벤트가 일어난 순간이다. 어떤 보내는 쪽은 바디 필드에 넣고 어떤 쪽은 서명에 곁들인다. 2장에서 자세히 보겠지만 지금은 "시간이 편지에 실려 온다"는 점만 알아 두자.

1.3 웹후크의 네 가지 약속

웹후크는 API 호출이 아니다. 웹후크는 그들이 보내고 내가 받는 관계다. 그리고 보내는 쪽은 내 통제가 닿지 않는다. 이게 웹후크 설계의 모든 어려움의 근원이다.

보내는 쪽의 서버는 느려질 수 있고 재시도할 수 있고 형식을 바꾸면서 아무에게도 말하지 않을 수 있다. 내 엔드포인트가 한 시간 동안 죽어 있으면, 그 사이에 쌓인 메시지가 돌아온 뒤 한꺼번에 쏟아진다. 그래서 수신자는 네 가지 약속을 지켜야 한다.

1. 믿는다. 이 편지를 보낸 자가 누구인지 증명할 수 있어야 한다. 공개된 URL로는 아무나 POST할 수 있다. 보내는 이를 확인할 수단이 필요하다.
2. 중복되지 않다. 같은 편지가 두 번 와도 괜찮아야 한다.
3. 잃지 않는다. 편지가 늦거나 순서가 뒤집혀도, 그 간극을 메울 수 있어야 한다.
4. 무너지지 않는다. 실패해도, 잃어버린 편지를 찾아 다시 전달할 길이 있어야 한다.

이 책은 그 순서로 진행한다. 다음 장은 약속 1, 그다음 장은 2와 3, 마지막 장은 4를 다룬다.

1.4 하나의 엔드포인트, 많은 이벤트 타입

대부분의 보내는 쪽은 웹후크를 하나의 URL로 몰아낸다. 이벤트 타입은 헤더나 바디의 필드에 실려 있다. Stripe는 수십 가지 이벤트 타입을, 내가

구독한 대로 같은 엔드포인트로 보낸다.

필요한 것만 구독해야 하는 이유는, 노이즈도 비용이기 때문이다. "API 키가 새로 생성됐다"는 이벤트를 받아도, 내 엔드포인트는 최소한 읽어야 하고 서명을 확인해야 하고 200을 돌려야 한다. 작지만 쌓인다. 더 중요한 건, 무엇을 할지 모르는 이벤트 타입을 받게 되면 "무시할지, 오류로 만들지"를 결정해야 한다는 점이다. 이 책의 답은 이것이다. 구독한 건 내 일이고 안 구독한 건 내 일이 아니다.

1.5 전달의 해부

하나의 전달을 쪼개 보자.

구성	역할	예
URL	수신 주소	/webhooks/stripe
method	전송 방식	POST
event type	무엇인지	charge.succeeded
event id	유일한 식별자	evt_
signature header	누가 보냈는지	Stripe-Signature
body	내용물	JSON

각 부분은 뒤에 나오는 장에서 따로 다룬다. 서명 헤더는 2장, event id는 3장, URL과 응답은 4장이다.

1.6 첫날: 서랍 만들기

웹후크를 실제로 세운다면, 순서는 이렇다.

먼저, 보내는 쪽 대시보드에서 웹후크 엔드포인트를 만들고 내 URL을 붙인다. GitHub는 Settings, Webhooks, Add webhook. Stripe는

Developers, Webhooks, Add endpoint. URL과 secret을 달라고 한다. secret은 편지에 서명할 때 쓰는 공유 키다. 내 서버 환경에 넣고 코드에 넣지 않는다.

둘째, 받을 이벤트를 고른다. GitHub는 repository 전체나 몇 개의 topic, Stripe는 실제로 쓰는 소수의 이벤트 타입.

셋째, 첫 전달을 확인한다. 엔드포인트를 만들면 보내는 쪽이 보통 테스트 이벤트를 하나 보낸다. 내 엔드포인트가 2xx를 돌리면 대시보드에 enabled 표시가 뜬다. 빨간 실패가 보이면, 에러를 어디서부터 읽어야 하는지 알 수 있다.

넷째, 로컬 개발이라면 공개 URL을 바로 줄 수 없다. 터널 도구(ngrok, cloudflared 등)가 임시로 로컬 포트를 밖으로 내어 준다. 아니면 클라우드에 임시 테스트 서버를 띄우는 것도 낫다.

첫날에 작은 팁을 하나 더. 엔드포인트가 새로 생겼을 때는 보내는 쪽의 전달 로그를 켜 두자. GitHub든 Stripe든 대시보드에 "언제 배달했고, 어떤 응답을 받았는지"를 남겨 준다. 뭔가가 꼬일 때, 내 서버 로그를 뒤지느니 이 쪽을 먼저 보는 게 빠르다.

1.7 코드 전에 가져가야 할 전제

가장 중요한 건 설계에 가져가는 전제다.

웹후크는 통신문이다. 보내는 쪽은 "한 번만, 순서대로"를 보장하지 않는다. 중복으로 올 수도 있고 순서가 뒤집힐 수도 있고 한동안 멈췄다가 몰려올 수도 있다. "정확히 한 번만 올 거야"라는 전제로 설계하면, 재시도가 처음 발생하는 순간 시스템이 깨진다. "여러 번 올 수 있고 순서가 어긋날 수 있고 안 올 수도 있다"는 전제로 설계해야 한다.

두 번째 전제: 바디는 진실이 아니다. 웹후크 바디의 데이터는 보낸 시점의

스냅샷이고, 내가 처리할 때는 이미 바뀔 수 있다. 안전한 답은 이벤트를 "무언가 일어난 신호"로 쓰고, 현재 상태는 보내는 쪽 API로 확인하는 것이다. 3장에서 이 기술을 본다.

세 번째: 내 엔드포인트의 응답이 프로토콜의 일부라는 점이다. 200인지 400인지 500인지에 따라 보내는 쪽의 행동이 달라진다. 언제 무엇을 답하는가는 실제 효과 있는 결정이다. 4장에서 다룬다.

1.8 첫주: 택배원이 실제로 한 일

이 장을 읽는 동안, 웹후크를 켠 뒤 실제로 벌어지는 첫주 이야기를 하나 그려 보자. 이게 이 책의 나머지 부분이 왜 필요한지를 보여 주기 때문이다.

첫날. 테스트 이벤트가 도착하고 대시보드가 초록색이 된다. 여기까지가 대부분 개발자의 기억에 남는 "웹후크 설정"이다.

둘째 날. 첫 실전 이벤트가 온다. 그리고 서명 검증이 실패한다. 이유는 secret을 두 개 준비해야 한다는 걸 모르고 있었기 때문이다. Stripe처럼 테스트 모드와 실전 모드를 나누는 서비스는, 각 모드마다 secret을 따로 발급한다. 테스트 secret으로 실전 이벤트를 검증하면, 아무런 경고 없이 서명이 안 맞는 것이다.

셋째 날. 실전 결제가 일어난다. 내 엔드포인트가 처리했고 라이선스를 발급했다. 5초 뒤, 같은 ID의 같은 이벤트가 또 온다. 보내는 쪽의 재전송이다. 내 쪽엔 중복 검사표가 없으므로, 두 번째 이벤트도 그대로 처리된다. 고객은 라이선스 메일을 두 번 받는다. 이번엔 큰일이 아니지만 "결제 완료" 대신 "환불 완료"가 두 번 왔다면 이야기는 다르다.

다섯째 날. 서버를 재배포한다. 재시작되는 10분 동안 엔드포인트는 죽어 있다. 돌아온 뒤, 쌓여 있던 세 이벤트가 한꺼번에 도착한다. 그중 하나는 순서가 뒤집혀서, 환불이 결제보다 먼저 처리된다. 내 상태 기계는 "환불할 결제가 없는데?" 하고 멈춘다.

이 네 번의 일은 모두 “이상”이 아니다. 보내는 쪽의 정상 동작이고 내 쪽의 준비 부족이다. 웹후크의 세계는 이렇게 동작한다. 첫주는 대부분 이 네 사건을 경험하게 되어 있다. 차이를 만드는 건 이 장에서 세운 전제와 다음 장들에서 세울 장치들이다.

1.9 이 책의 지도

1인 개발자에게 운영 부담이 진짜 비용이다. on-call 팀은 없다. 새벽에 고장을 알아채는 사람도 나뿐이다. 그래서 이 책은 같은 웅덩이에 두 번 빠지지 않게 하는 매뉴얼이다.

2장은 약속 1을 세운다. 편지가 진짜라는 걸 어떻게 아느냐. 3장은 약속 2와 3. 두 번 오는 편지, 순서가 어긋난 편지를 어떻게 받느냐. 4장은 약속 4. 편지를 잃어버렸을 때, 어떻게 다시 찾아오느냐. 각 장 끝에는 체크리스트가 있다. 엔드포인트를 진짜 세상에 열기 전에, 하나씩 확인하는 용도다.

첫 줄의 코드를 쓰기 전에, 보내는 쪽의 웹후크 문서를 한 번 읽어라. 어떤 헤더가 있고 어떤 이벤트 타입이 있으며 재시도 정책이 무엇인지. 읽을 때 네 가지를 물어라. 이건 진짜 보내는 쪽의 것인지 어떻게 알지? 하나의 전달을 유일하게 식별하는 ID는 뭐지? 내가 답을 안 하면 그들은 뭘 하지? 내가 500을 답하면 그들은 뭘 하지?

이 네 가지를 끝내면, 첫날은 끝이다. 다음 장에서는 약속 1, 보내는 이를 확인하는 부분부터 세운다.

2 보내는 이를 믿는 법: 서명 검증과 재생 공격

2.1 누가 보냈는지 모르면 열지 마라

약속 1, 믿는 일이다.

웹후크 URL이 새면 어떤 일이 일어나는지 생각해 보자. URL은 서버 로그에 남고 벤더 설정 화면에 뜨고 프록시에서 보일 수 있다. URL을 아는 사람은 누구든 "결제 완료" 메시지를 만들어 올 수 있다. 내 엔드포인트는 그 편지가 진짜 Stripe 것인지 알 방법이 없다. 위조자는 JSON 모양만 복사하면 된다. 바디를 그대로 처리한다면, 위조자는 내 제품을 무료 구입한 셈이다.

그러니 1차 방어선은 보내는 이 확인이다.

IP 허용 목록은 충분한 답이 아니다. 보내는 쪽은 IP 범위가 여러 개이고 예고 없이 바꾼다. 목록을 하드코딩하면, 그쪽이 바꾼 날 내 엔드포인트가 죽는다. 보안 계층으로서도 신뢰할 수 없는 이유가 있다. 허용 목록에 없는 IP에서 온 건 전부 버리는데, 보내는 쪽이 새 범위로 이주한 날에는 실전 이벤트가 전부 버려진다.

진짜 답은 서명이다. 나와 보내는 쪽은 secret이라는 키를 공유한다. 보내는 쪽은 이 키로 메시지에 대해 해시를 계산해 헤더에 실어 보낸다. 내 쪽은 같은 secret으로 같은 해시를 다시 계산하고 둘이 맞으면 "이 편지는 secret을 아는 자가 쓴 것"이다. secret은 네트워크를 지나가지 않는다. 지나가는 건 secret의 "효과"뿐이다. 그래서 위조가 비싸진다. 알고리즘은 대개 HMAC-SHA256이다.

2.2 왜 평범한 해시 concat이 아닌가

"secret을 바디 뒤에 붙여 sha256을 돌리면 되는데?"라는 유혹이 있다. 안 된다. 해시 함수의 내부 구조를 아는 공격자는, 원래 해시 값과 secret을 몰면서 바디 뒤에 임의의 데이터를 이어 붙인 새 메시지의 해시를 만들어낼 수 있다. 이런 length extension 취약점이 일부 해시 구조에 실제로 존재한다.

HMAC은 이 문제를 구조적으로 회피한다. secret을 바깥에 그냥 붙이는 대신, 해시를 두 번 돌리면서 secret을 두 갈래로 뿌린다. 그래서 "원본 해시만 있으면 더 길게 위조할 수 있다"는 공격이 통하지 않는다. 표준 알고리즘이 존재할 때, 스스로 해시 기반 인증을 발명할 이유도, 그럴 능력도 없다. 표준을 써라.

실무적으로는 한 줄로 정리된다. 서명 = HMAC-SHA256(secret, canonical payload). canonical payload, 즉 서명에 들어갈 문자열이 무엇인지는 보내는 쪽 문서를 확인하라. 바디 그 자체인 경우도 있고 GitHub처럼 sha256= 접두어가 붙는 경우도 있고, Stripe처럼 타임스탬프가 앞에 오는 경우도 있다. 이 장의 코드가 그 세 가지 모양을 보여 준다.

2.3 서명이 실제로 생기는 모양

보내는 쪽마다 모양이 다르지만 뼈대는 같다. 서명은 헤더로 온다.

GitHub는 raw 바디의 sha256 HMAC을 X-Hub-Signature-256 헤더에 실어 준다. 값은 sha256=로 시작하는 16진수 문자열이다. Stripe는 한 뉘앙스 있다. Stripe-Signature 헤더에 t(타임스탬프)와 v1(서명) 두 부분을 쉼표로 구분해 넣고, 서명의 입력은 바디만이 아니라 "t + 마침표 + 바디"라는 문자열이다. 이 차이가 뒤에 큰 효과가 있으니, 코드와 함께 보자.

```
import hashlib
import hmac
import time

def verify_github(raw_body: bytes, header: str, secret: bytes) ->
↪  bool:
 expected = hmac.new(secret, raw_body,
↪  hashlib.sha256).hexdigest()
 return hmac.compare_digest(f"sha256={expected}", header)

def verify_stripe(raw_body: bytes, header: str, secret: bytes,
 tolerance: int = 300) -> bool:
 parts = dict(p.split("=", 1) for p in header.split(","))
 t = parts.get("t", "")
 v1 = parts.get("v1", "")
 if not t.isdigit() or abs(time.time() - int(t)) > tolerance:
 return False
 signed = t.encode() + b"." + raw_body
 expected = hmac.new(secret, signed, hashlib.sha256).hexdigest()
 return hmac.compare_digest(expected, v1)
```

여기서 볼 것이 세 가지다.

첫째, raw_body. 서명은 보낸 그대로의 바이트에 대해 계산된다. 프레임워크가 바디를 읽어서 JSON으로 파싱하고 내가 다시 직렬화하면, 공백 하나나 키 순서 하나가 해시를 바꾼다. 서명이 영영 안 맞는, 로컬에서는 되고 프로덕션에서만 깨지는 그 고전적 버그의 정체가 여기 있다. 대부분의 프레임워크는 request.get_data() 같은 방법으로 raw 바디를 건넨다. 서명 검증을 raw 바디에 대해 하라. 파싱은 그 다음 일이다.

둘째, compare_digest. 두 문자열을 ==로 비교하면, 맨 앞 글자가 안 맞으면 일찍 끝난다. 비교에 걸린 시간이 몇 글자까지 맞았느냐에 따라

달라지고 공격자는 응답 시간을 재며 해시를 한 바이트씩 맞출 수 있다. compare_digest는 항상 같은 시간에 끝난다. 과장처럼 보이지만, 이건 표준이다.

셋째, Stripe의 tolerance. 타임스탬프가 서명에 참여한다. 5분 이상 지난 편지는 무효다. 이게 다음 공격을 막는 장치다.

2.4 재생 공격: 진짜 편지의 위험

replay, 재생 공격은 이렇게 한다. 공격자가 위조를 하지는 않는다. 진짜 편지를 훔친다. 내 서버 로그, 프록시, 모니터링 도구, 그 어디든 유효한 서명과 바디가 남아 있을 수 있다. 그걸 나중에 내 엔드포인트에 다시 보내면, 서명 검증은 그대로 통과한다. 진짜 편지이니까. 다만 배달이 늦었을 뿐이다.

재생 공격에는 두 겹의 방어가 있다.

첫째 겹은 시간이다. Stripe처럼 타임스탬프가 서명에 들어 있으면, 정해진 시간 이전의 편지는 무효로 만들 수 있다. GitHub 헤더에는 타임스탬프가 없으므로, 이 겹은 얇다.

둘째 겹은 중복 검사다. 재생된 편지에는 event id가 있다. 그 id를 이미 처리했다면, 이 편지는 낡은 것이라고 판단해 버릴 수 있다. 결국 재생 방어도 3장의 중복 테이블에 기대는 셈이다. 서명은 “이건 진짜냐”에 답하고 중복표는 “이건 새것이냐”에 답한다. 둘 다 필요하다.

이게 왜 “편지는 여러 번 온다”고 전제해야 하는지 보여 주는 예다. 재생 공격은 이국적인 예외가 아니라, 의도가 있는 중복이다.

2.5 검증 실패의 응답: 무엇을 돌려주나

검증이 실패한 편지는 절대로 처리해선 안 된다. 그리고 보내는 쪽이 영원히 재시도하도록 만들지 않는 방법으로 답해야 한다. 401이나 400이다. "열 수 없는 편지다"라는 답이지, "바빠"라는 답이 아니다. 위조된 메시지에 500을 돌려주면, 보내는 쪽은 재시도를 계속하고 내 로그는 쓰레기로 가득 찬다.

로그에 대한 주의도 여기서 다룬다. 검증 실패 시, 출처 IP, 시각, 이벤트 타입(읽힐 때), 어떤 체크에서 떨어졌는지를 남긴다. secret 자체는 남기지 않는다. 서명 헤더 전체를 그대로 dump하지도 않는다. secret이 로그에 새는 순간, 약속 1은 무너진다.

2.6 갑자기 서명이 안 맞을 때

서명 검증이 프로덕션에서 갑자기 죽는 경우를 네 가지만 알아 두면, 대부분은 현장을 벗어날 수 있다.

첫째, secret 불일치. 테스트 모드 secret으로 실전 이벤트를 검증하는 것이다. 1장에서 그린 둘째 날의 일이다. 두 모드가 있는 서비스라면, secret도 두 개임을 확인하라.

둘째, 바디가 바뀌었다. 프록시나 로드 밸런서에서 Content-Type에 charset을 붙이거나, 바디를 재단장하는 경우가 있다. 서명은 보낸 그대로의 바이트에 대해 계산되므로, 중간에서 한 바이트만 바뀌어도 안 맞는다. 확인 방법은 간단하다. 보내는 쪽 대시보드의 전달 로그에 남아 있는 raw 바디와, 내가 받은 바디를 byte 단위로 비교한다.

셋째, 프레임워크 버전 차이. 같은 raw 바디라도, 프레임워크가 건네 주는 형태가 바뀐다. 업그레이드 뒤 서명이 죽었다면, 우선 request.get_data() 결과부터 의심하라.

넷째, 로컬과 프로덕션의 URL이 달라서, 실전 이벤트가 로컬 터널로 가고 있었다. 서명이 아니라 도착지가 바뀌은 경우다.

네 가지 모두, 답은 같다. 보내는 쪽 대시보드의 전달 로그에서 "그쪽이 보낸 그대로"를 확보하고, 내 쪽이 받은 바와 byte 단위로 비교하는 것이다. 서명 버그는 추론으로 풀지 못한다. 바이트의 차이가 전부다.

2.7 secret의 관리와 교체

secret은 나와 보내는 쪽이 공유하는, 아는 자가 편지를 위조할 수 있는 값이다. 데이터베이스 비밀번호와 같은 등급으로 취급한다.

코드에 쓰지 말고 환경 변수나 secret store에 넣는다. git에 커밋해 버린 경우는 고전 중의 고전이다. 이미 커밋했다면, 교체 절차는 이렇다.

1. 보내는 쪽 대시보드에서 새 secret을 만든다.
2. 테스트 이벤트를 보내, 새 서명이 나에게까지 오는지 확인한다.
3. 코드에 구, 신 두 secret으로 모두 검증하는 로직을 배포한다. 한 번에 전환 가능하다면, 코드부터 바꾸고 대시보드에서 전환하는 쪽도 가능하다.
4. 며칠간 안정을 본 뒤, 보내는 쪽에서 구 secret을 삭제한다.

두 secret의 병행 창이 필요한 이유는 겹치는 순간 때문이다. 코드를 배포하기 전에 구 secret을 지워 버리면, 그 사이 실전 편지가 죽는다. 배포하고 전환하면, 차분하게 바꿀 수 있다.

교체는 응급 절차가 아니라 정기 작업이다. 몇 개월마다, 아니면 secret을 알던 개발자가 나갔을 때 돌린다. 대시보드 조작은 1분이고 덕분에 전체 사슬이 살아 있는지 한 번 확인할 수 있다.

2.8 서명을 안 주는 보내는 쪽이 있을 때

Stripe, GitHub, 그리고 대부분의 알려진 서비스는 서명을 제공한다. 하지만 직접 연동하는 작은 서비스, 내부 도구, 혹은 오래된 시스템은 서명 헤더를 아예 안 주는 경우도 있다.

이 경우에도 아무 검증 없이 받으면 안 된다. 쓸 수 있는 장치를 조합하라.

첫째, URL에 일회성 토큰을 붙인다. https://example.com/hooks?token=abc123 식이다. 토큰을 아는 자만 엔드포인트를 찾는다. 이건 비공개성을 키우는 장치이지, 암호화는 아니다. 로그와 브라우저 히스토리에 URL이 남는다는 걸 기억하라.

둘째, 출처를 확인한다. 보내는 쪽의 IP가 고정되어 있다면 허용 목록을 쓴다. 단, 1장에서 말한 대로 목록은 주기적으로 대조할 것. 셋째, 그리고 이건 필수다. 중복 테이블과 이벤트 id를 그대로 쓴다. 서명이 없으면 "이건 진짜냐"에 답할 장치가 약해지므로, "이건 새것이냐"에 답할 장치를 강하게 만들라는 의미다.

서명이 없는 연동은 항상 "임시"로 취급하라. 대시보드에 한 줄 남겨 두자. "이 엔드포인트는 서명 없이 토큰만 검증한다. 서비스쪽이 서명을 지원하면 전환할 것." 임시라는 메모가 없으면, 임시 장치가 영구 장치가 된다.

2.9 이번 장의 체크리스트

엔드포인트를 열기 전에, 하나씩 확인한다.

- 서명 검증은 raw 바디에 대해 하는가? 재직렬화한 값이 아닌가?
- 문자열 비교는 compare_digest를 쓰는가?
- 타임스탬프 허용이나 중복 id 체크가 있는가?
- 검증 실패 시 4xx를 돌려주는가?
- secret은 환경에 있고 코드와 git에는 없는가?

- 교체 절차를 문서에 적어 두었는가?

다 체크했다면, 약속 1은 끝이다. 편지 상자는 열려 있지만 열쇠는 내 손에 있다. 다음 장에서는 약속 2와 3을 다룬다. 두 번 오는 편지, 순서가 뒤집힌 편지를 어떻게 받는지.

3 중복과 순서를 다스리는 법: 멱등 키와 상태 재조정

3.1 중복은 예외가 아니라 정상이다

2장 끝에서 재생 공격은 의도가 있는 중복이라고 말했다. 여기서 한 걸음 더 강하게 말하자. 웹후크 수신자에게 중복은 예외가 아니라 정상이다.

이유는 보내는 쪽의 재시도 방식에 있다. 보내는 쪽은 편지를 보낸 뒤, 정해진 시간 안에 2xx를 받지 못하면 다시 보낸다. 이제 이런 상황을 그려 보자. 내 엔드포인트가 편지를 받고 처리를 시작했고 성공했다. 하지만 응답 패킷이 네트워크에서 사라졌거나, 처리가 늦어져 보내는 쪽의 타이머가 먼저 다 됐다. 보내는 쪽 입장에선 "답이 없었으니, 다시 보낸다". 내 입장에선 "이미 처리했다". 같은 편지가 두 번 온다. 그리고 이 두 번 모두 합법적인 동작이다. 어느 쪽도 바보가 아니다.

이걸 at-least-once delivery라고 부른다. 보내는 쪽은 "최소 한 번"을 약속할 뿐, "정확히 한 번"은 약속하지 않는다. 그리고 순서도 약속하지 않는다. 재시도가 있고 워커 서버가 여러 대면, 같은 고객의 편지가 순서 없이 도착할 수 있다.

그래서 내 엔드포인트는 두 조건을 만족해야 한다. 같은 편지를 두 번 처리해도 결과가 같아야 하고, 늦게 온 편지나 순서가 뒤집힌 편지가 상태를 깨지 않아야 한다. 이 장은 바로 그 두 조건을 세운다.

3.2 중복 테이블: 1차 방어선

가장 단순한 구조는, 어떤 event id를 이미 받았는지 기록하는 테이블이다.

```
CREATE TABLE webhook_events (
 event_id TEXT PRIMARY KEY,
 received_at TIMESTAMPTZ NOT NULL DEFAULT now(),
 status TEXT NOT NULL DEFAULT 'pending'
);
```

편지가 도착하면 event id를 테이블에 넣는다. insert가 성공하면 처음 온 것이다. 중복 키로 실패하면 이미 받은 것이다.

```
def receive(event_id: str) -> bool:
 try:
 conn.execute(
 "INSERT INTO webhook_events(event_id) VALUES (%s)",
 (event_id,)
 )
 return True
 except IntegrityError:
 return False
```

중복 편지에도 2xx를 답해야 한다. 보내는 쪽은 2xx를 받을 때까지 재시도를 멈추지 않는다. 중복에 400을 답하면, 보내는 쪽은 중복을 계속 보내고 내 로그는 중복으로 가득 찬다. 중복 테이블은 바로 그 루프를 끊는 장치다.

status 필드는 다음 단계의 힌트다. "받았지만 아직 처리하지 않았다"는 상태가 존재한다. 언제 그런 일이 일어나나? 예를 들어, 편지를 받고 200을 먼저 돌렸고 실제 처리는 나중에 백그라운드에서 하는 구조다. 또는 편지를 받은 뒤, 처리하기 전에 서버가 죽은 경우. 이 경우 테이블이 내 큐가 된다. 시작할 때 status가 pending인 편지를 다시 읽어 처리하면 된다.

1인 개발자에게 이 책이 권하는 패턴이다. 진짜 메시지 큐를 도입할 필요가 없다. 데이터베이스의 테이블이면 된다. 중요한 건 200을 돌려주기 전에

이벤트를 테이블에 쓰는 일이다. 처리를 먼저 하고 쓰기를 나중에 하면, 그 사이 죽는 순간 편지를 잃고 잃었는지조차 모른다.

3.2.1 테이블이 오래되면

중복 테이블은 영원히 키면 안 된다. event id가 매일 수천 개씩 쌓이면, 몇 년 뒤 테이블은 쓸모없는 돌탑이 된다. 그리고 돌탑도 문제를 만든다. 인덱스 조회가 느려지고 백업이 커진다.

해법은 단순하다. received_at 기준의 보존 기간을 정한다. 30일, 90일, 보내는 쪽의 최대 재전송 기간보다 넉넉히. 그 뒤의 행은 지운다. 지우는 작업은 cron 하나면 충분하다.

여기 한 가지 함정이 있다. 보존 기간이 보내는 쪽의 재전송 기간보다 짧으면 안 된다. 만약 Stripe가 3일까지 재전송하는데, 내가 1일 뒤의 중복 기록을 지워 버리면, 3일 차에 오는 재전송은 "처음 온 편지"로 취급된다. 중복 테이블의 보존 기간은, 보내는 쪽 문서의 재전송 기간을 확인한 뒤 정해야 한다.

3.3 "하자"에서 "이다"로: 멱등한 로직

중복 테이블은 같은 편지가 다시 오는 경우만 막아 준다. "다른 id로 온 같은 사실"은 막지 못한다.

예를 들어 보자. Stripe가 "payment succeeded" 이벤트를 보냈다. 라이선스를 발급했다. 얼마 뒤, 같은 결제에 대한 "invoice paid" 이벤트가 온다. 사실은 하나인데, 이벤트 타입은 다르다. id가 다르므로 중복 테이블은 막아 주지 못한다. 로직이 "라이선스를 한 개 추가한다"면, 두 번 추가된다. 로직이 "이 고객의 상태를 licensed로 설정한다"면, 몇 번을 돌려도 결과는 하나다.

이게 연산과 상태의 차이다. "쿠폰을 더한다"는 연산이고 두 번 하면 두 개가

된다. “쿠폰 적용 플래그를 세팅한다”는 상태이고, 몇 번을 해도 결과는 같다. 비즈니스 로직은 가능하면 “상태를 세팅하는” 형태로 쓰자.

실무 기준을 하나 준다. 처리 전에 “이미 참인 것이 무엇이냐”를 먼저 본다. 결제 성공 이벤트가 오면, “이 고객은 이미 이 플랜의 라이선스를 가진가?”를 먼저 확인한다. 있으면 추가하지 말고 만료일만 갱신한다. 이런 구조에서는 중복이 어떤 경로로 와도 해가 없다. 중복 테이블은 1차 방어선이고 멱등 로직은 마지막 방어선이다. 1차가 새면, 마지막이 잡아야 한다.

두 방어선이 같은 일을 하는 것처럼 보이지만 역할은 다르다. 중복 테이블은 “이 id는 봤다”고 말한다. 멱등 로직은 “이 결과는 이미 세워져 있다”고 말한다. id가 바뀌면 전자는 무력해지지만 후자는 여전히 일한다.

3.4 순서를 믿지 마라: 상태 기계

다음 문제다. 같은 고객의 이벤트가 순서 없이 오면, 어떻게 되는가.

예를 들어 보자. “결제 완료”가 오면 라이선스를 발급한다. “환불”이 오면 발급을 취소한다. 이게 정상이다. 그런데 서버가 잠깐 죽어 있어서 “환불”이 먼저 도착하면? 아직 발급하지도 않은 결제를 환불하려 하고, 실패한다. 그다음에 “결제 완료”가 오면, 라이선스를 발급한다. 최종 상태는 “발급됨”인데, 고객은 환불했다. 돈이 새는 구조다.

답은, 수신자가 “올바른 순서를 기다리는” 것이 아니다. 올바른 순서를 내가 모른다. 답은 상태 기계다. 상태를 명시적으로 정의하고 합법적인 이동만 허용하며 합법적이지 않은 이동은 “알 수 없다”로 처리한다.

```
created -> paid -> refunded
```

각 이동에는 전제가 있다. refunded는 현재 상태가 paid일 때만 의미가 있다. created 상태에서 refunded 이벤트가 오면, “나는 이 결제의 역사를 놓쳤다”는 뜻이다. 이때 추측하지 말자. 보내는 쪽 API에 “이 결제의 현재

상태가 뭐냐"고 물어라.

이게 상태 재조정, reconciliation이다. 1장에서 암시한 기술이다. 웹후크 바디는 스냅샷이고 API의 답은 현재다. 이벤트 id가 "이 엔티티에 일이 있었다"고 알려 주고, API가 "지금은 이 상태다"고 알려 준다. 둘을 합치면 강해진다. 늦게 온 이벤트, 순서가 뒤집힌 이벤트, 내 쪽이 죽어 있던 동안 일어난 이벤트. 전부 API로 물어보면 메워진다.

보내는 쪽이 엔티티마다 sequence number를 제공한다면, 처리한 마지막 sequence와 비교해 순서 역전을 감지할 수 있다. 하지만 모든 보내는 쪽이 제공하는 건 아니므로, 기본값은 재조정이다. sequence가 있으면 쓰고 없으면 물어본다.

상태 기계를 코드로 하나 적어 둔다. 짧을수록 좋다는 뜻에서다.

```
TRANSITIONS = {
 ("created", "paid"): "paid",
 ("paid", "refunded"): "refunded",
}

def apply(current: str, event: str) -> str:
 next_state = TRANSITIONS.get((current, event))
 if next_state is None:
 reconcile_from_api() # 모르는 이동: 추측 대신 확인
 return current
 return next_state
```

이 코드에서 주목할 것은 if 분기다. 모르는 이동이 오면 예외를 던지지 않고 API로 재조정한다. 예외를 던지면 그 편지는 데드레터로 가며, 재조정이라는 더 나은 길을 걷지 못한다. 상태 기계는 "모를 때의 행동"까지 포함해야 완성이다.

3.5 200은 빨리: 시간 규율

한 가지 더. 내 엔드포인트는 얼마나 빨리 답해야 하는가. 가능한 한 빨리. 1초 안쪽이 좋다.

이유는 장 시작의 재시도 논리다. 보내는 쪽은 타이머를 갖고 있다. 타이머가 다 되기 전에 2xx를 받지 못하면, 실패로 판단해 다시 보낸다. HTTP 요청 안에서 결제 API 호출과 라이선스 발급까지 하면, 수 초가 걸린다. 느린 편지가 모두 중복이 된다.

그래서 받고 처리하는 것을 분리한다. 핸들러의 일은 세 가지만이다. 서명을 확인하고 테이블에 쓰고 200을 돌려 준다. 진짜 일은 뒤에 하는 무엇인가가 한다. 1인 개발자에게 그 "무엇"은 같은 프로세스의 백그라운드 스레드, 혹은 주기적으로 pending 테이블을 읽는 cron이 될 수 있다. 작은 시스템에서는 데이터베이스 테이블 자체가 큐가 된다.

인프로세스 스레드의 위험을 하나 더 말해 둔다. 서버가 재시작되면, 스레드에 떠 있던 편지는 사라진다. 그래서 테이블의 pending 패턴이 필수다. 테이블에 있으면, 재시작이 두렵지 않다. 시작할 때 다시 읽으면 된다.

받기 먼저, 처리 나중이라는 구조에는 한 틈이 있다. 테이블에 썼는데, status를 processed로 바꾸기 전에 죽으면, 다음에 pending으로 다시 읽고 중복 처리를 한다. 멱등 로직이 있어서 괜찮다. 하지만 "괜찮다"는 전제를 확인해야 한다. 처리 로직이 멱등하지 않은 채 pending 재읽기만 하면, 재시작이 중복 사고를 만든다. 순서를 기억하자. 테이블 쓰기, 200, 처리, status 갱신. 이 네 가지를 다 세우고 그 다음에 처리 로직이 멱등한지 확인한다. 둘 중 하나만 하면 안 된다.

3.6 세 가지 패턴, 한눈에

상황	대응	결과
같은 id 재도달	중복 테이블에 이미 있음	200 반환, 무시
다른 id 같은 사실	상태 확인 후 갱신	멱등하게 수렴
순서 뒤집힘	API로 현재 상태 동기화	상태 기계로 복구

3.7 이번 장의 체크리스트

- 200을 돌려주기 전에, event id를 테이블에 쓰는가?
- 중복 편지에도 2xx를 답하는가?
- 비즈니스 로직이 “상태를 세팅”하는 것인가, “연산을 수행”하는 것인가?
- 상태 기계가 불가능한 이동을 받으면, API로 물어 보는가?
- 핸들러는 1초 안에 끝나는가?
- 중복 테이블의 보존 기간이 보내는 쪽 재전송 기간보다 긴가?

약속 2와 3은 끝이다. 편지 상자에는 중복 필터가 있고 책상에는 상태 기계가 있다. 다음 장에서는 마지막 약속, 편지를 잃어버렸을 때와 보내는 쪽이 계속 두드릴 때의 일이다.

4 실패를 운영으로 삼는 법: 재전송, 데드레터, 재플레이

4.1 내 응답은 명령이다

3장 끝에서 "무엇을 답할지"를 남겨 두었다. 여기서부터다.

엔드포인트의 응답 코드는 보내는 쪽에게 내리는 명령이다. 2xx는 "받았다, 다시 보낼 필요 없다"다. 4xx는 "이 편지가 잘못됐다, 다시 보내지 마라"다. 5xx는 "내 쪽이 고장 났다, 나중에 다시 보내라"다. 타임아웃은 5xx와 같다, 다시 보내라는 뜻이다.

여기서 힘이 이중으로 작용한다. 잘 쓰면 보내는 쪽의 재시도가 내 안전망이 된다. 잘못 쓰면 편지를 잃거나, 끝없는 재전송 루프에 빠진다.

첫째 규칙: "못한다"와 "안 한다"를 구분한다.

- 안 한다: 페이로드 형식이 틀리고 이벤트 타입을 모르며 검증이 실패한 경우. 재시도가 내용을 바꾸지 못한다. 4xx.
- 못한다: 데이터베이스가 죽어 있고 하류 API가 타임아웃되고 버그가 예외를 던진 경우. 재시도가 성공할 가능성이 있다. 5xx.

가장 위험한 것은, "못한다"에 200을 돌려주는 일이다. 보내는 쪽은 배달 완료로 여겨 다시 보내지 않는다. 편지는 조용히 사라진다. 두 번째로 위험한 것은, "안 한다"에 500을 돌려주는 일이다. 보내는 쪽은 영영 성공하지 못할 편지를 계속 재전송하고, 내 로그와 중복 테이블이 쓰레기로 가득 찬다.

예상하지 못한 이벤트 타입이 오면, 뭘 답할까. 내가 정말로 구독한 타입인데 핸들러를 아직 못 만들었다면, 500을 답한다. 이 고장을

알아채고 싶으니까. 내가 안 구독한 타입이 온다면(보내는 쪽 설정 실수, 혹은 형식 변경), 200을 답하되, 시끄럽게 로그를 남긴다. 차이는 이것이다. 전자는 내가 고칠 bug이고 후자는 바깥세계에서 온 소문이다.

4.2 재전송 스케줄과, 내게 주어진 시간

보내는 쪽마다 재시도 정책은 다르지만 모양은 비슷하다. 몇 초 뒤, 몇 분 뒤, 몇 시간 뒤, 간격을 넓히며 몇 번 재시도한다. 횟수와 총 기간은 서비스마다 다르니 문서에서 확인하라. 대개 하루 이틀을 넘기지는 않는다 [추정].

포인트는, 내게 시간이 주어진다는 사실이다. 데이터베이스가 20분 죽어 있어도, 편지가 사라지지 않는다. 보내는 쪽이 보관하고 계속 두드린다. 이건 웹후크가 내 쪽의 내부 API 호출과 가장 많이 다른 지점이다. 재전송 시스템을 온전히 내 손으로 지어 올 필요가 없다.

하지만 거꾸로도 성립한다. 그 시간 안에, 버그가 내 핸들러에 있으면, 모든 재시도가 같은 벽에 부딪힌다. 코드의 한 줄이 틀리면, 그 타입의 모든 편지가 실패한다. 그래서 "빨리 고친다"는 운영 규율이다. 실패가 반복되면, 문제는 네트워크가 아니라 코드다.

4.3 편지 모양이 바뀔 때

보내는 쪽은 형식을 바꾼다. 필명을 바꾸고 필드를 빼고 새 필드를 넣는다. 예고 없이, 혹은 예고하되 내 쪽을 기다려 주지 않고.

대응은 세 가지다.

첫째, 필수 필드만 믿고 나머지는 무시하라. 내 처리에 꼭 필요한 필드가 없으면 그건 4xx의 정당한 사유다(형식이 틀린 편지니까). 없지만 해롭지는 않은 필드가 생겼다면, 알아채고 무시한다. JSON을 파싱할 때 "모르는 키"가 오면 예외를 던지는 방식은, 보내는 쪽의 형식 변경 하나에 전체가

죽는 방식이다.

둘째, 버전을 확인하라. 어떤 보내는 쪽은 헤더나 바디에 형식 버전을 실어 준다. 있다면, 지원 범위를 명시하고 범위를 벗어난 버전은 4xx로 답하되 시끄럽게 로그를 남긴다.

셋째, 모양 변경은 실패가 아니라 신호다. 필드가 사라진 건 형식 변경일 수 있고, 내 쪽의 버그일 수도 있다. 둘을 구분하는 법은, 보내는 쪽의 전달 로그에 남아 있는 raw 바디를 보는 것이다. 그쪽이 실제로 보낸 바디에 필드가 없으면 형식 변경, 있으면 내 파싱의 버그다.

4.4 내 쪽의 데드레터: 보내는 쪽 주차장만 믿지 마라

보내는 쪽 대시보드의 주차장은 보조다. 기초가 될 수 없다. 유효 기간이 지나면 사라질 수 있고 형식이 바뀔 수 있으며, 잃어버린 편지를 찾으려고 벤더 콘솔에 로그인하는 구조는 안 된다.

그래서 내 쪽에 기록을 둔다. 원본 편지와, 그 편지에 무슨 일이 있었는지를.

```
CREATE TABLE webhook_log (
 id BIGSERIAL PRIMARY KEY,
 event_id TEXT,
 raw_body BYTEA,
 headers TEXT,
 status TEXT,
 err TEXT,
 received_at TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE dead_letters (
 event_id TEXT PRIMARY KEY,
 reason TEXT,
```

```
 died_at TIMESTAMPTZ DEFAULT now(),
 replayed_at TIMESTAMPTZ
);
```

raw body를 쓰는 일이 왜 중요한가. 로그에 남은 바디는 "내 코드가 파싱하고 다시 직렬화한" 값이므로, 보내는 쪽이 실제로 보낸 것과 다를 수 있다. raw 바이트만이, 같은 서명 검증으로 재실행할 수 있는 유일한 출처다. headers는 secret을 제외한 채로 저장하면, 서명 헤더도 함께 남길 수 있다.

편지가 언제 죽는가? 이 책의 단순한 기준: 처리 실패 3회, 혹은 24시간 이상 미처리. 테이블을 점검해 이 조건에 해당하는 행을 dead_letters로 옮기는 cron이면 된다. 그 순간, 전화로 알림을 보낸다. 1인 개발자에게 알림은 나 자신에게 오는 문자다. 채널이 중요한 게 아니라, "나는 알아채는가"다.

4.5 잃어버린 편지를 되돌리기

데드레터를 발견했다면, 복구 절차는 고정되어 있다. 루틴으로 만들자.

1. 버그를 고친다. 고장 난 핸들러에서 그대로 재전송하면, 데드레터만 더 생긴다.
2. 알려진 정상 이벤트로 몸을 데운다. 보내는 쪽의 "테스트 이벤트 전송" 버튼, 혹은 저장해 둔 샘플. 파이프라인이 살아 있는지 확인한다.
3. 원래 순서대로 재전송한다. 여러 개면 received_at 순서대로.
4. 중복은 이미 안전하다. 3장의 중복 테이블과 멱등 로직 덕분에, 재전송의 반복은 해가 없다.
5. API로 간극을 메운다. 다운이 길었으면, 보내는 쪽이 전달하지도 못했고 id조차 모르는 이벤트가 있다. 3장의 재조정: 중요한 엔티티(최근 고객, 진행 중인 주문)에 대해 API로 현재 상태를 물어본다.

재전송은 "HTTP 요청을 다시 보내는 것"이 아니다. "상태를 확인하고 순서대로, 안전하게"라는 전체 절차다. raw body를 저장해 두면, 이 절차는 10분짜리 작업이다.

4.6 모니터링: 세 가지 숫자

운영에서 볼 것은 무엇이냐. 1인 개발자에게 세 개면 충분하다.

1. 실패. 최근 1시간의 5xx와 예외 수. 0이면 좋고 연속이면 코드에 문제가 있다.
2. 정체. 가장 오래된 pending 행의 received_at에서 지금까지의 경과. 편지가 처리되지 않고 앉아 있으면, 뭔가 막혀 있다.
3. 침묵. 마지막 편지가 도착한 지부터의 시간. 이건 미묘하다. 웹후크 스트림에는 기준선이 있다. 항상 편지가 오던 결제 서비스에서 갑자기 조용해지면, 이상 신호다. URL 변경, secret 교체, 보내는 쪽 사고. "오지 않는 것"도 이벤트다.

세 숫자를 확인하고 전화로 알림을 보내는 cron 스크립트 하나면 된다. 대시보드를 지어 올 필요 없다. 모양은 대개 이렇다.

```
def watch():
 fails = count_last_hour("webhook_log", status_in=("error",
↪  "dead"))
 stuck = oldest_pending_age("webhook_events")
 quiet = hours_since_last("webhook_log")
 if fails > 3 or stuck > timedelta(hours=1) or quiet > baseline *
 ↪  3:
 page_me()
```

baseline은 내 시스템의 평소 침묵 기간이다. 결제가 항상 일어나는 프로덕션이라면 1시간, 드문 이벤트라면 12시간. 이 상수를 대충 잡으면,

침묵 경고가 매번 거짓 경보가 된다. 거짓 경보 세 번이면, 사람은 경보에 반응하지 않는다.

4.7 테스트: 세 단계

URL을 진짜 세상에 넘기기 전에, 세 단계로 테스트한다.

1. 형식 테스트. 페이로드를 직접 만들어, 내 secret으로 서명하고 로컬 엔드포인트에 POST한다. raw 바디, 헤더 파싱, 서명 비교의 버그를 잡는다. 20줄짜리 스크립트면 된다.
2. 보내는 쪽 테스트. 대시보드의 "테스트 이벤트 전송"을 쓴다. secret, URL, 이벤트 구독이 제대로 맞물리는지 확인한다.
3. 실패 테스트. 의도적으로 500을 한 번 답하고 보내는 쪽이 재전송하는지, 두 번째를 중복 테이블이 잡는지 확인한다. 이게 약속 2와 4를 함께 검증하는 테스트다. 출시 전에 하라. 사고 뒤에 하면 안 된다.

3단계는 흔히 건너뛴다. 하지만 이 단계가 전체 설계를 증명하는 곳이다. 테스트에서 중복이 안 보이면, 보내는 쪽이 재전송하지 않는 것이거나(문서 확인), 중복 검사가 안 되는 것이거나(버그). 둘 중 하나다.

4.8 고전 8개 함정

1인 개발자가 실제로 넘어지는 순서로 적는다.

1. 재직렬화한 바디로 서명 검증. 로컬에서는 되고 프로덕션에서만 깨진다.
2. 핸들러 안에서 동기 처리. 느린 편지가 중복이 되고 중복이 버그가 된다.
3. 중복 테이블 부재. 재시도 한 번이 이중 발급이 된다.
4. 내부 오류에 400. 편지는 재시도 없이 죽는다.

5. secret 교체 계획 부재. 새는 secret은 전체를 무너뜨린다.
6. 순서 가정. 상태 기계가 거꾸로 간다.
7. API 확인 부재. 스냅샷만 믿다, 간극에 빠진다.
8. 프로덕션의 URL이나 secret 불일치. 엔드포인트는 살아 있지만 편지는 옛 주소로 간다.

각 함정의 대응은 앞 장에 있다. 목록의 역할은, 출시 전에 하나씩 확인하게 하는 것이다.

4.9 런북: 전화가 울렸을 때

이 책의 마지막 산출물은 코드가 아니라 문서다. 새벽에 반쯤 깨어 있는 상태로 따라갈 수 있는 1페이지 런북.

- 실패 알림이면, webhook_log의 마지막 에러를 확인하고, 죽은 이벤트 타입을 찾는다.
- 데드레터면, 5단계 재전송 절차를 따른다.
- 침묵이면, 보내는 쪽 대시보드의 상태를 확인하고, URL과 secret의 일치 대조를 한다.
- 중복 폭풍이면, 먼저 중복 테이블을 본다. 테이블에 있으면, 200이 정상이다.

런북이 파일이라면, 저장소에 넣는다. 내 머릿속이라면, 런북이 아니다.

4.10 마무리

네 가지 약속. 믿고 중복되지 않게 하고 잃지 않고 무너지지 않게. 웹후크는 통신문이고 편지 상자는 내 시스템의 경계다. 그 경계를 넘는 모든 것은, 확인되고 기록되고 중복 제거되고 재실행 가능해야 한다.

오늘 하나를 구현한다면, 둘부터 시작하라. raw 바디에 대한 서명 검증,

그리고 200 이전의 중복 테이블. 이 둘이, 첫 재시도에 깨지는 시스템과 재시도를 흘려보내는 시스템의 차이다. 나머지 전부다.