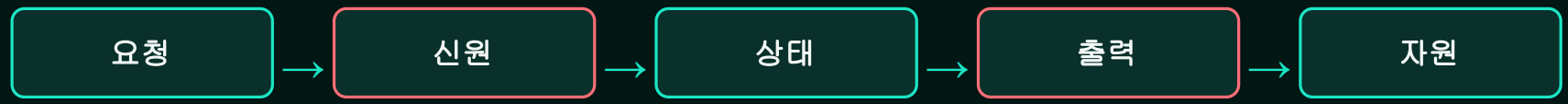
Secure by Design

안전한 웹 백엔드의
진화와 트레이드오프

구성 한눈에 보기

10개 섹션 흐름으로 보는 전체 구성



기본 프레임

모든 기술 선택을 입력·상태·출력·자원 네 칸으로 본다

입력(Input)

상태(State)

출력(Output)

자원(Resource)

취약점 이름보다 먼저 볼 것: 요청의 종류

요청 의도와 시스템 해석 방식이 위험을 만든다

정상처럼 보이는 요청

01
필수값 누락

02
특수 파라미터 조합

03
경계값 초과

의도적 공격 요청

01
SQLi·경로조작

02
토큰 탈취 시도

03
권한 우회

입력 → 해석기 → 권한 → 출력

서버 요청

사용자는 UI를 누르지만 공격자는 HTTP 자체를 보낸다

실수 섞인 정상 요청	`GET /posts?page=-1&status=unknown`	특수 파라미터 · 경계값 → 논리 오류	→	기능 버그 / 검증 누락
경계값 / 자원 고갈형	`POST /search {"q":"A" x 10MB}`	긴 문자열 · 거대한 body → CPU/메모리 폭증	→	타임아웃 / 비용 폭탄
악의적 요청	`POST /login {"id":"OR 1=1"}`	SQLi · 경로 조작 · XSS → 명시적 공격	→	무결성 훼손 / 계정 탈취
정상 응답이지만 과한 노출	`GET /users/124 → {role,phone,note}`	내부 필드 · 과도한 관계 로딩	→	개인정보 노출 / BOLA

정상 요청이지만 실수로 사고가 나는 경우

01

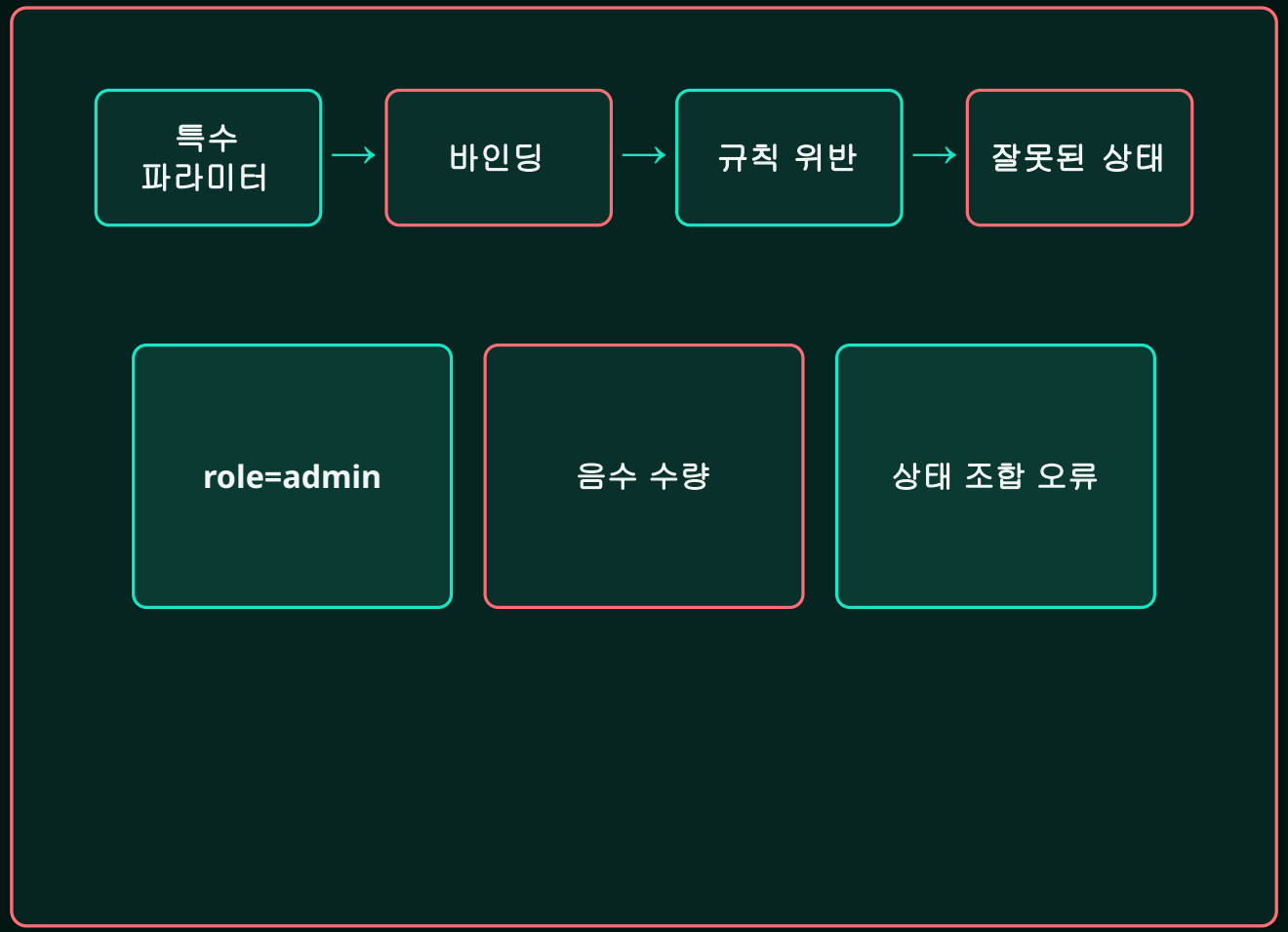
예: role=admin 같은 숨은 필드가 바인딩되어 논리 오류가 발생

02

예: 금액, 수량, 상태값의 조합이 비즈니스 규칙을 깨뜨림

03

예: 삭제·수정 요청에서 본인 확인 없이 ID만 믿고 수행



긴 문자열과 극단 입력은 요즘에도 위험하다

01

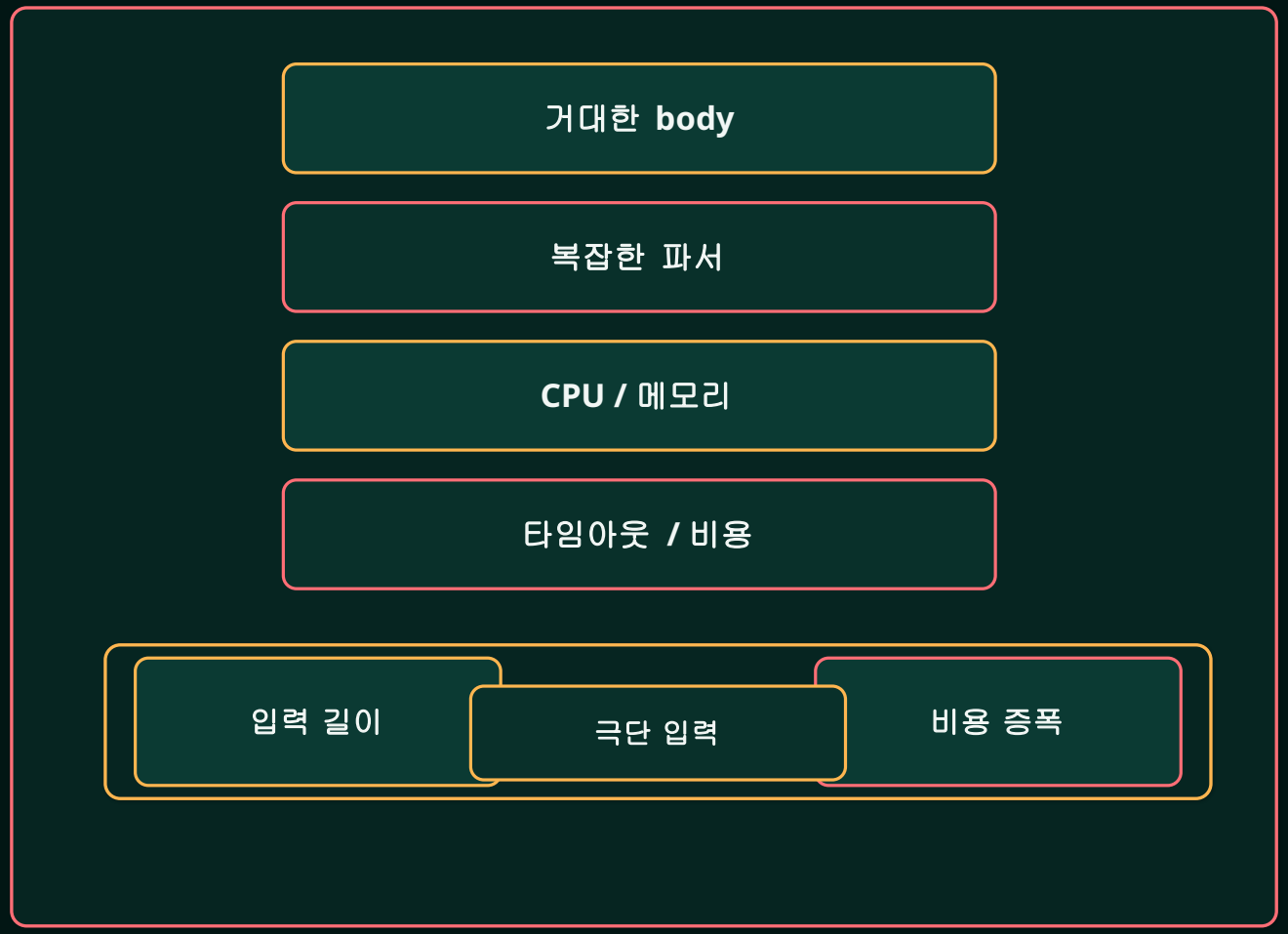
메모리 안전 언어라도 대용량 JSON·압축 해제·정규식은 CPU/메모리를 태운다

02

길이 제한이 없으면 DB 인덱스, 로그, 메시지 큐, 캐시 키 설계까지 흔들린다

03

'오버플로우'가 아니어도 할당 폭증, 타임아웃, 비용 증폭으로 서비스가 무너진다



악의적인 요청: SQLi는 대표 예시일 뿐이다

01

SQL Injection: 쿼리와 데이터가 섞이면 명령으로 해석된다

02

Path Traversal/SSRF: 서버가 대신 파일·URL에 접근하도록 속일 수 있다

03

Mass Assignment: 바인딩 가능한 필드가 곧 권한 우회 통로가 된다

과도한 요청: DDoS는 네트워크만의 문제가 아니다

01

공격자는 많은 요청을 보낼 수도 있고, 적은 요청으로 비싼 연산을 반복시킬 수도 있다

02

로그인, 검색, 내보내기, 이미지 변환, 외부 API 호출은 모두 비용 표면이다

03

가용성 문제는 곧 보안 문제이며, 예산·SLA 문제이기도 하다

Request storm

IP / body limit

Gateway rate limit

App timeout

DB 보호

정상 요청에도 과도한 정보 노출이 생긴다

01

본인 프로필 조회 API가 내부 컬럼까지 그대로 노출하는 경우

02

다른 사용자 ID로 접근했을 때 소유권 검사 없이 응답하는 경우

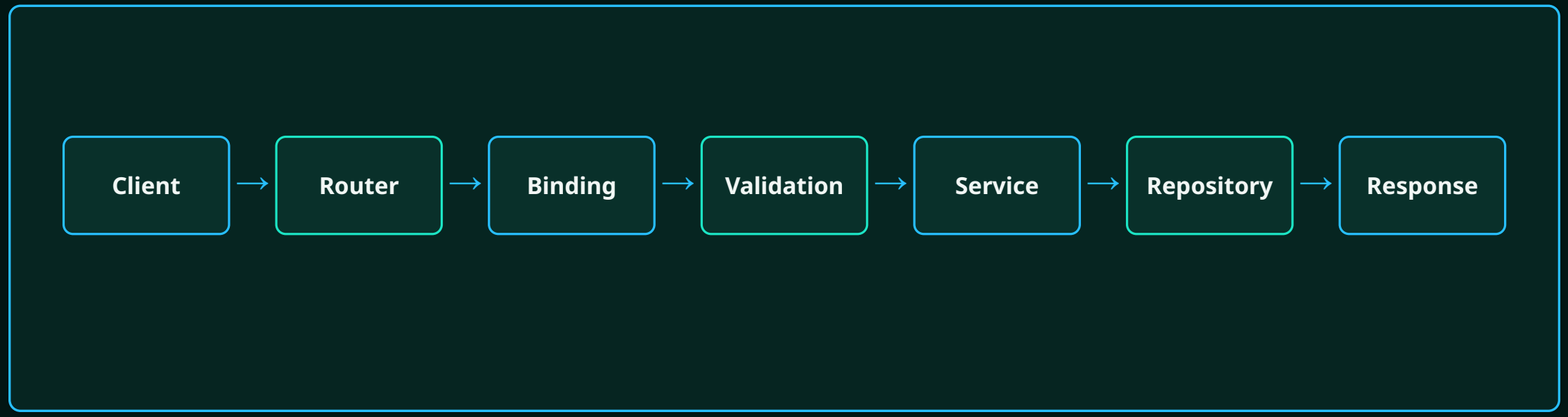
03

목록 API가 필요 이상의 필드를 기본 응답으로 반환하는 경우

필드	응답 처리
id	노출
name	노출
phone	상황별
role	최소화
internal_note	비노출

HTTP 요청 파이프라인 마다의 역할

Go I/O 경계

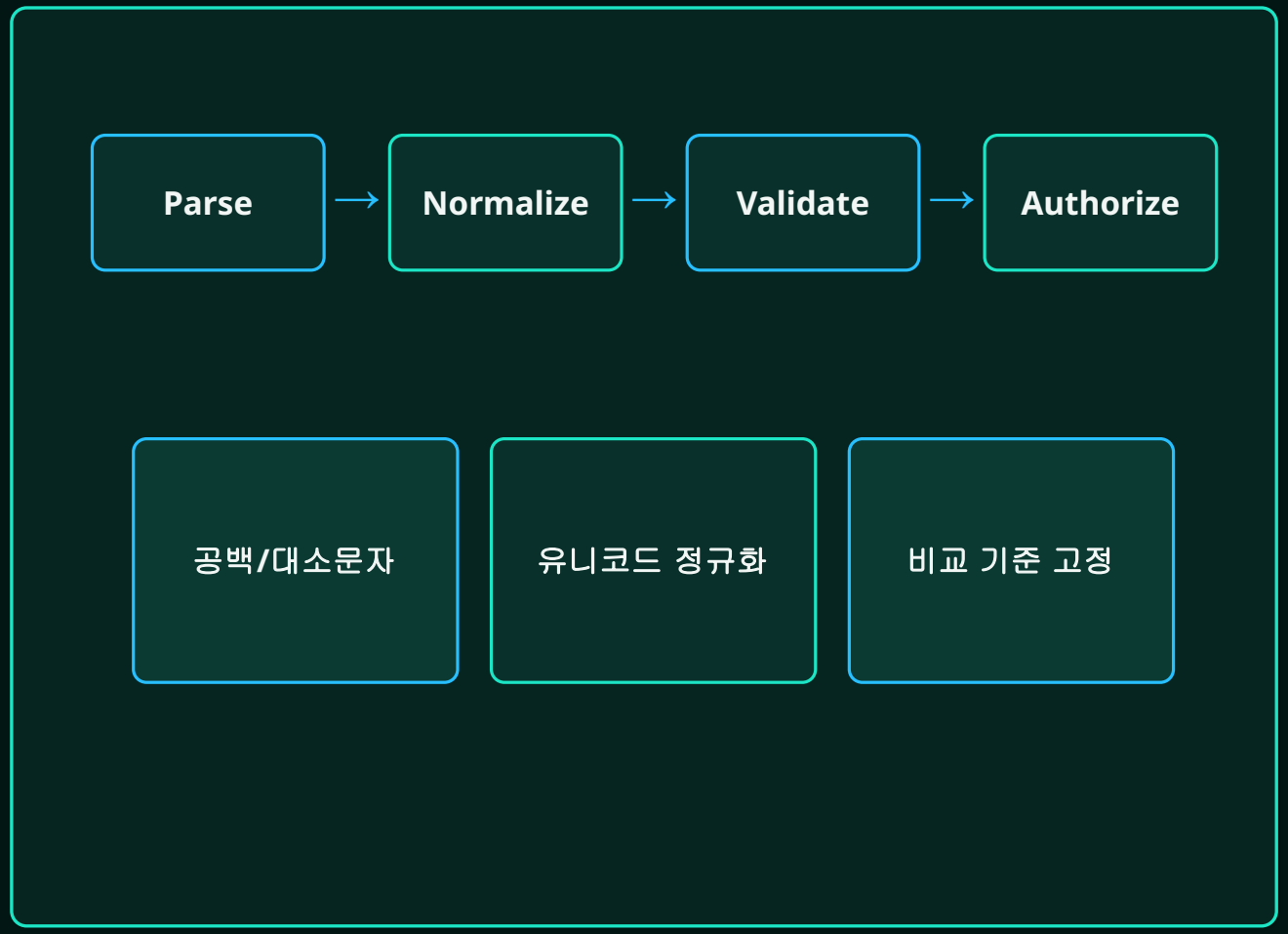


신뢰 경계와 정규화(canonicalization)를 먼저 잡아라

01
동일한 의미의 값도 인코딩·공백·대소문자·유니코드 형태가 다를 수 있다

02
'비교' 전에 표준 형식으로 정규화하지 않으면 우회가 생긴다

03
검증 순서는 parse → normalize → validate → authorize 가 안전하다



구조적 검증과 의미 검증은 다르다

Go I/O 경계

01

구조적 검증: 필수값, 길이, 형식, enum, 범위

02

의미 검증: 현재 상태에서 가능한지, 소유권이 맞는지, 재고가 있는지

03

구조가 맞아도 비즈니스 규칙을 깨면 거부해야 한다

구조적 검증

01

필수값 / 범위 / 형식

02

파싱 입구에서 빠르게 차단

의미 검증

01

상태 전이 / 소유권

02

비즈니스 규칙과 권한

화이트리스트 검증이 기본이다

Go I/O 경계

01

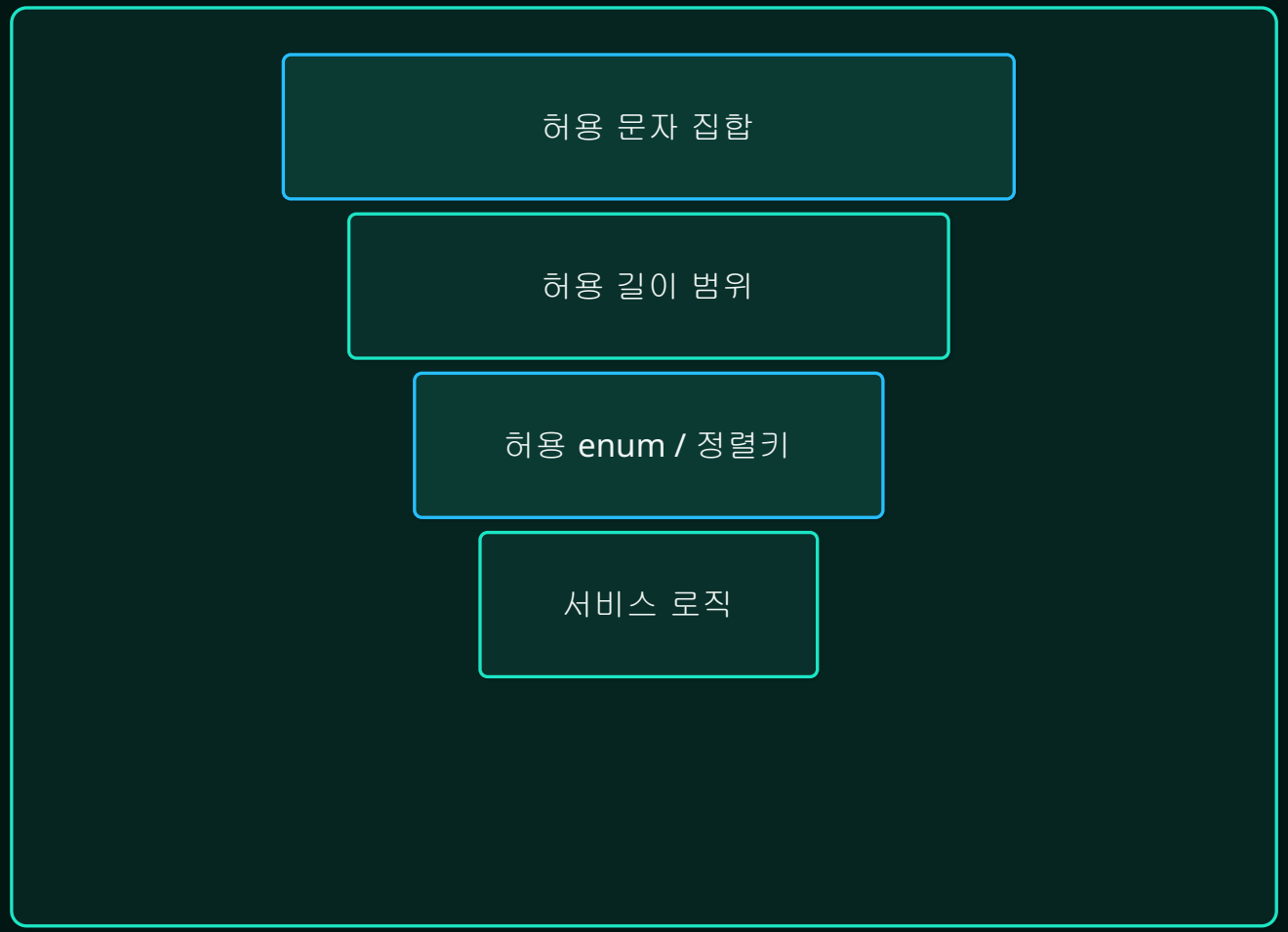
허용 가능한 문자, 길이, 값의 범위를 먼저 정의한다

02

블랙리스트는 우회 케이스가 끝없이 추가된다

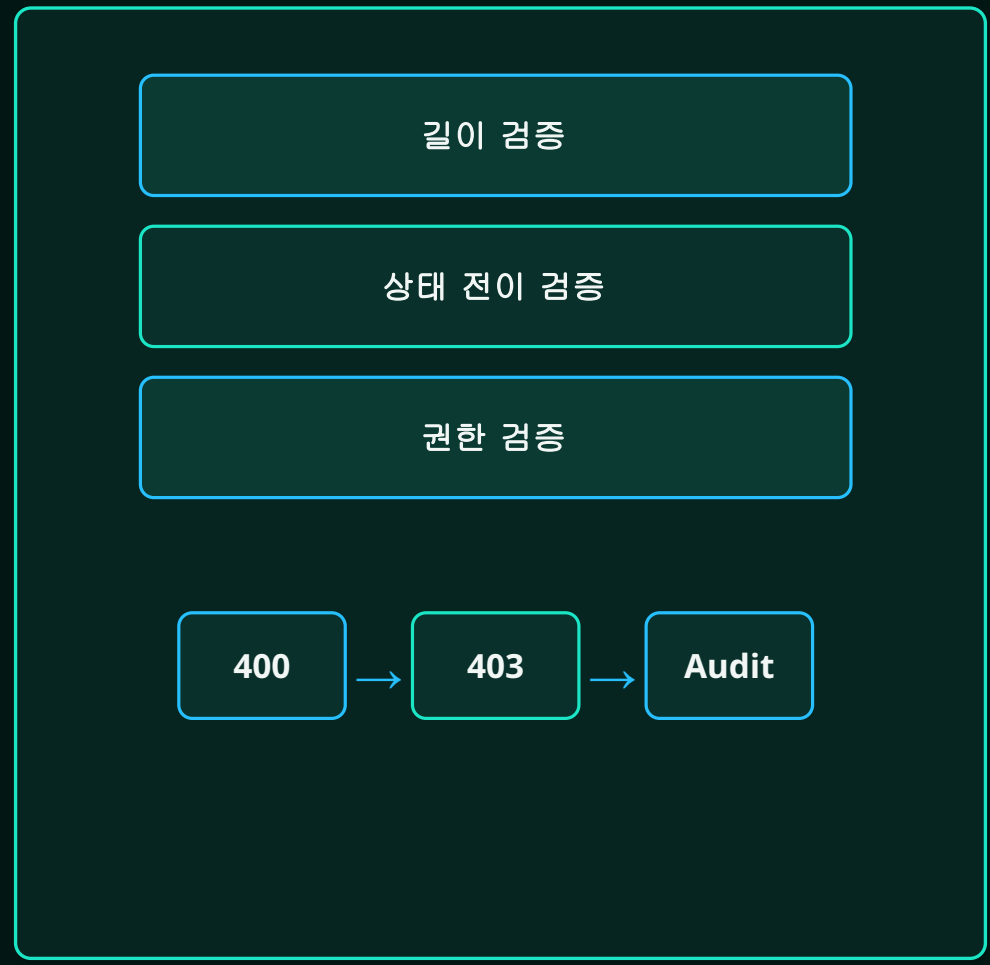
03

파일, URL, 정렬키, 필터 연산자처럼 해석기가 붙는 입력은 특히 좁혀야 한다



Go 검증 코드 예시: 길이·상태·권한을 분리한다


```go
if utf8.RuneCountInString(req.Title) > 100 {
    return badRequest("title too long")
}
if req.Status == "published" && !actor.CanPublish() {
    return forbidden("not allowed")
}
```



에러 메시지와 로그도 공격 표면이다

Go I/O 경계

01

사용자 응답에는 추상화된 메시지, 내부 로그에는 원인과 스택을 남긴다

02

민감정보·토큰·세션 ID는 로그에 남기지 않거나 마스킹한다

03

상관관계 ID(correlation ID)로 추적성을 확보하되 개인정보와 분리한다

출력도 검증 대상이다: DTO와 인코딩

Go I/O 경계

01

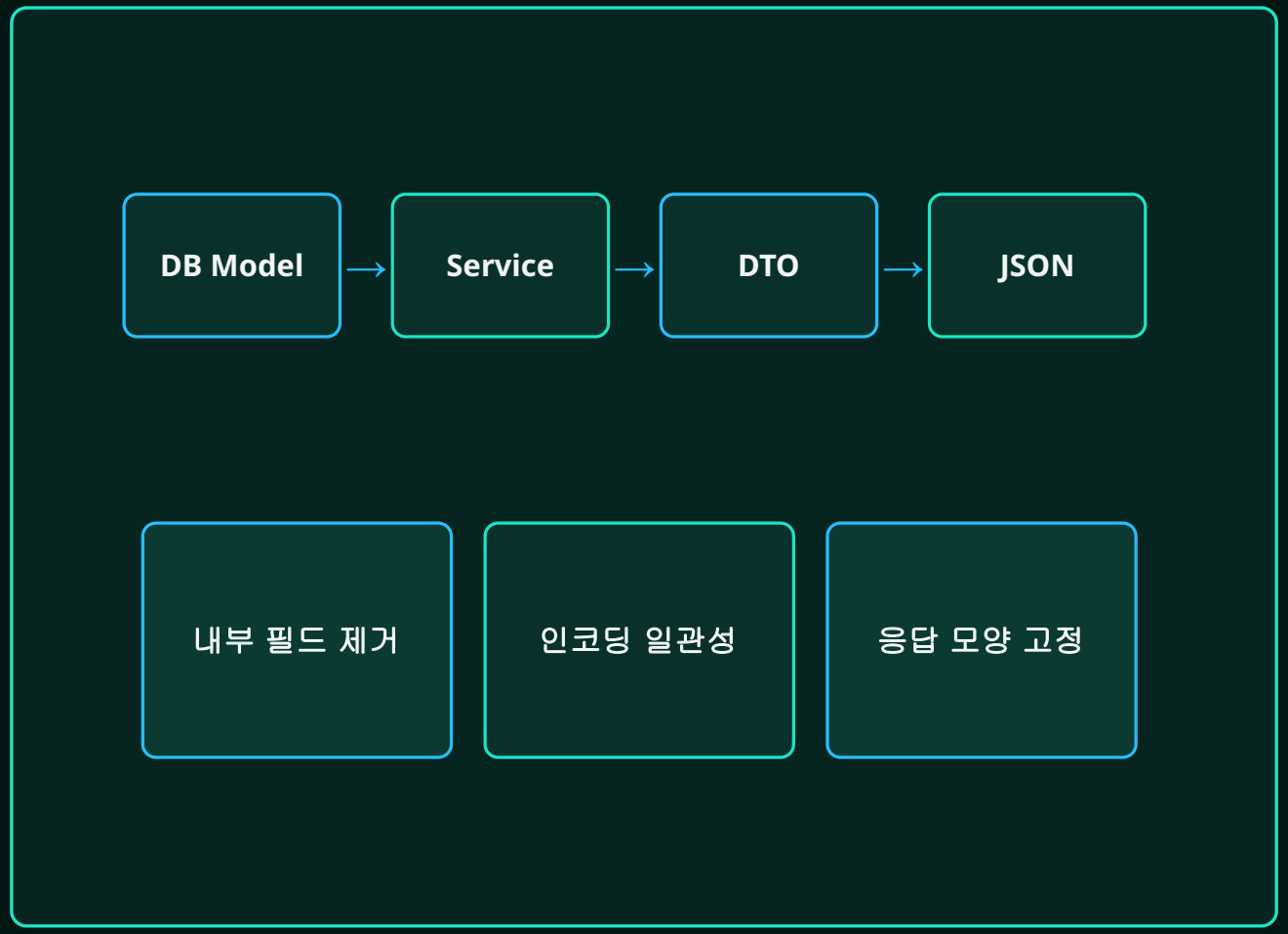
DB 모델을 그대로 JSON으로 내보내지 않는다

02

응답 DTO로 필드 화이트리스트를 만들고, 권한별 응답을 분리한다

03

브라우저 렌더링이 있으면 XSS 관점의 출력 인코딩 책임도 명확히 한다



SQL Injection의 본질: 데이터가 명령문 위치로 들어간다

01

문자열 연결은 입력을 SQL 문법 일부로 승격시킨다

02

정렬 컬럼·필터 연산자·Raw SQL도 같은 문제를 가진다

03

ORM을 써도 Raw Query나 동적 조합은 다시 위험해진다

문자열 결합

01

쿼리와 데이터가 섞임

02

명령문 위치 침범

인자 바인딩

01

구조 고정

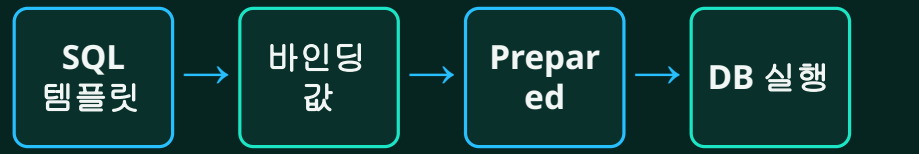
02

데이터는 값으로만 전달

방어 기본기: Prepared Statement와 좁은 쿼리 표면


```go
// 위험
db.Raw("SELECT * FROM posts WHERE id = " + id)

// 안전
db.Where("id = ?", id).First(&post)
```



- 쿼리 구조 고정
- 데이터는 명령어가 아님
- 권한 좁은 쿼리 표면

비밀번호 저장의 기본: 평문 저장 금지, 복호화도 금지

01

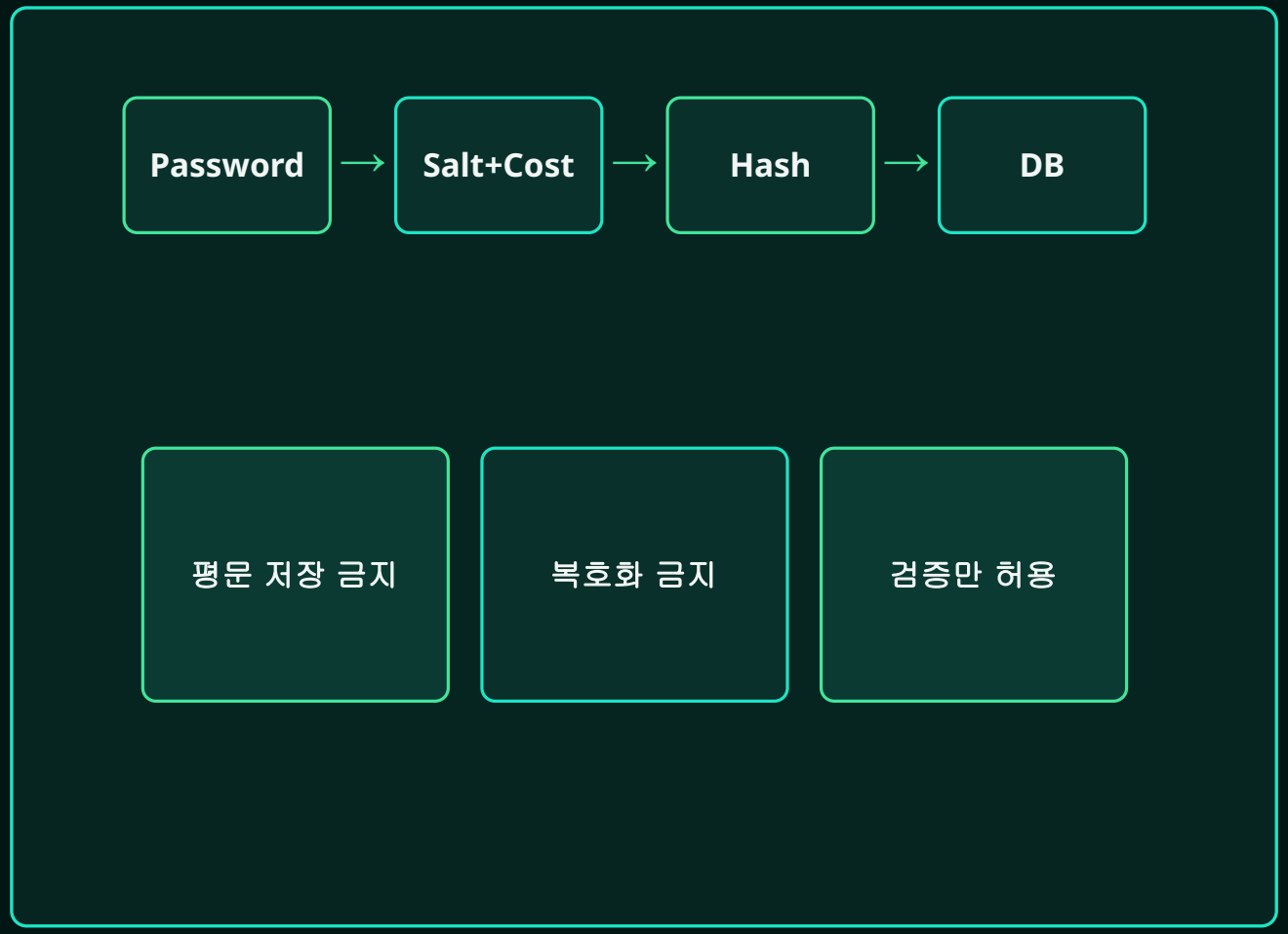
비밀번호는 대칭키로 '암호화'하는 것이 아니라 단방향 해시로 저장한다

02

유출을 전제로 설계하므로, DB만 털려도 즉시 재사용되기 어렵게 만들어야 한다

03

로그·예외·테스트 데이터에도 비밀번호가 남지 않게 한다



bcrypt/Argon2는 보안을 올리지만 비용도 올린다

01

의도적으로 느린 해시는 크리덴셜 스터핑·브루트포스를 어렵게 한다

02

비용 파라미터를 너무 높이면 로그인 자체가 DoS 표면이 된다

03

실무에서는 사용자 경험, 서버 비용, 공격 모델을 함께 본다

로그인 속도

cost factor

브루트포스 저항

보안↑ = CPU 비용↑

운영 환경별 튜닝 필요

과도하면 로그인 UX 저하

세션은 낡은 기술이 아니라 '상태를 서버가 기억하는 방식'이다

01

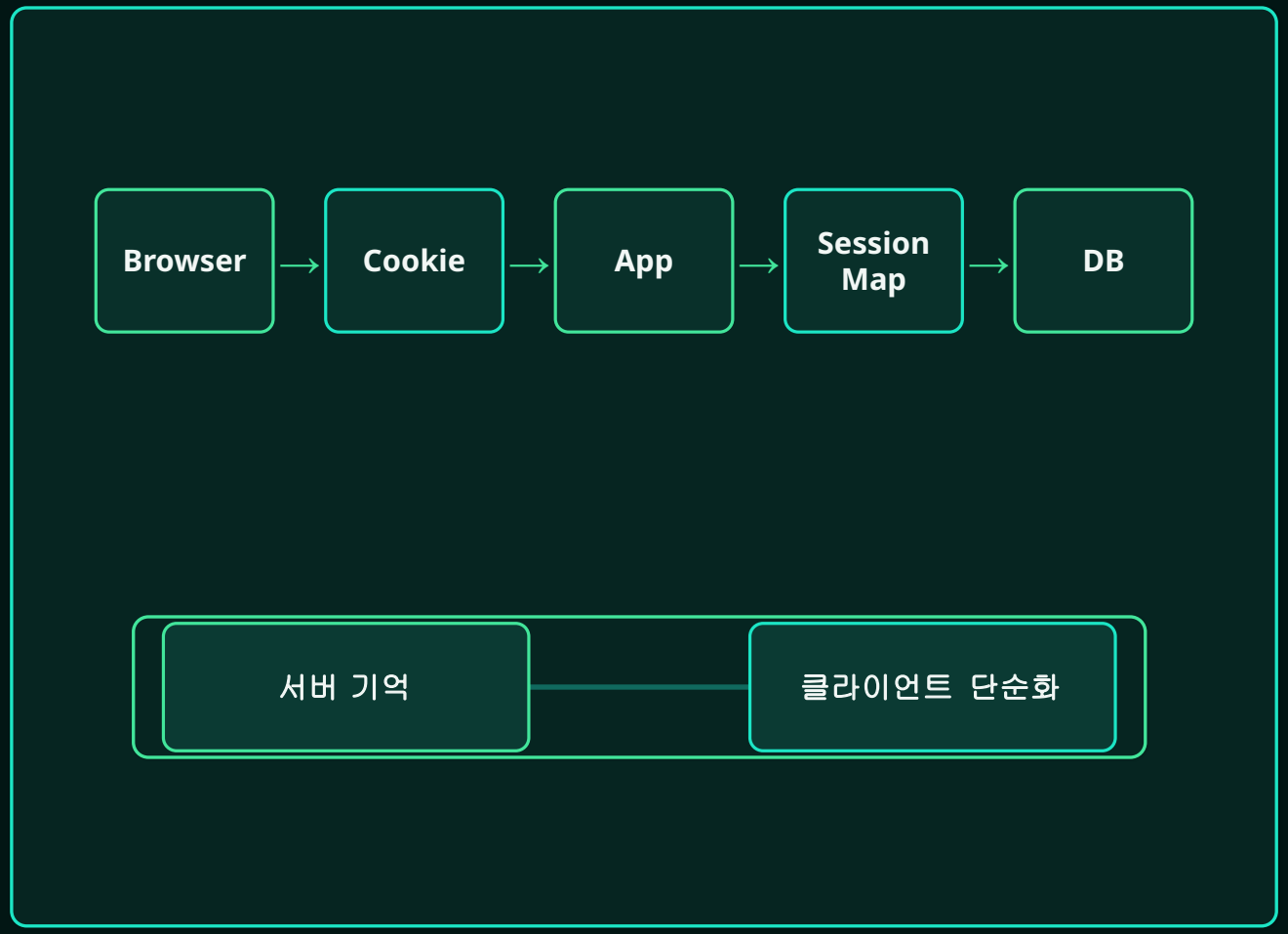
사용자가 로그인하면 서버는 세션 ID와 인증 상태를 연결해 둔다

02

클라이언트는 세션 ID만 들고 다니고, 실제 상태는 서버가 소유한다

03

즉시 만료, 강제 로그아웃, 권한 변경 반영이 직관적이다



쿠키 기반 인증은 CSRF를 반드시 같이 생각해야 한다

01

브라우저는 쿠키를 자동 전송하므로, 타 사이트 유도 요청이 성립할 수 있다

02

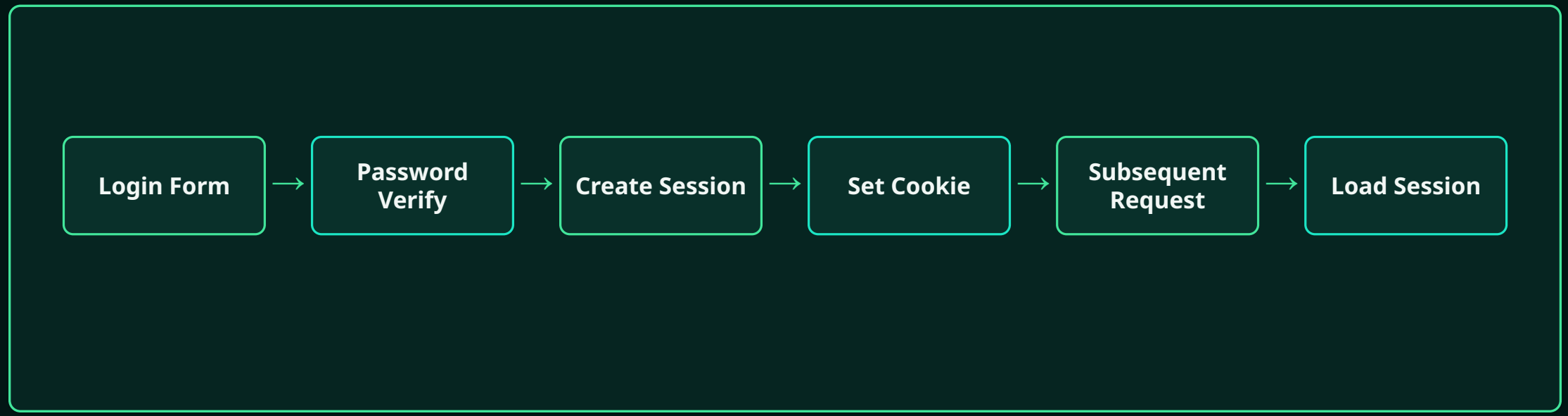
SameSite, CSRF token, 재인증, 민감 작업 분리 등의 조합이 필요하다

03

보안을 높일수록 UX와 클라이언트 구현 비용이 늘 수 있다

클래식 모놀리식 로그인 흐름

비밀번호와 세션



과거에 이렇게 개발한 이유

비밀번호와 세션

01

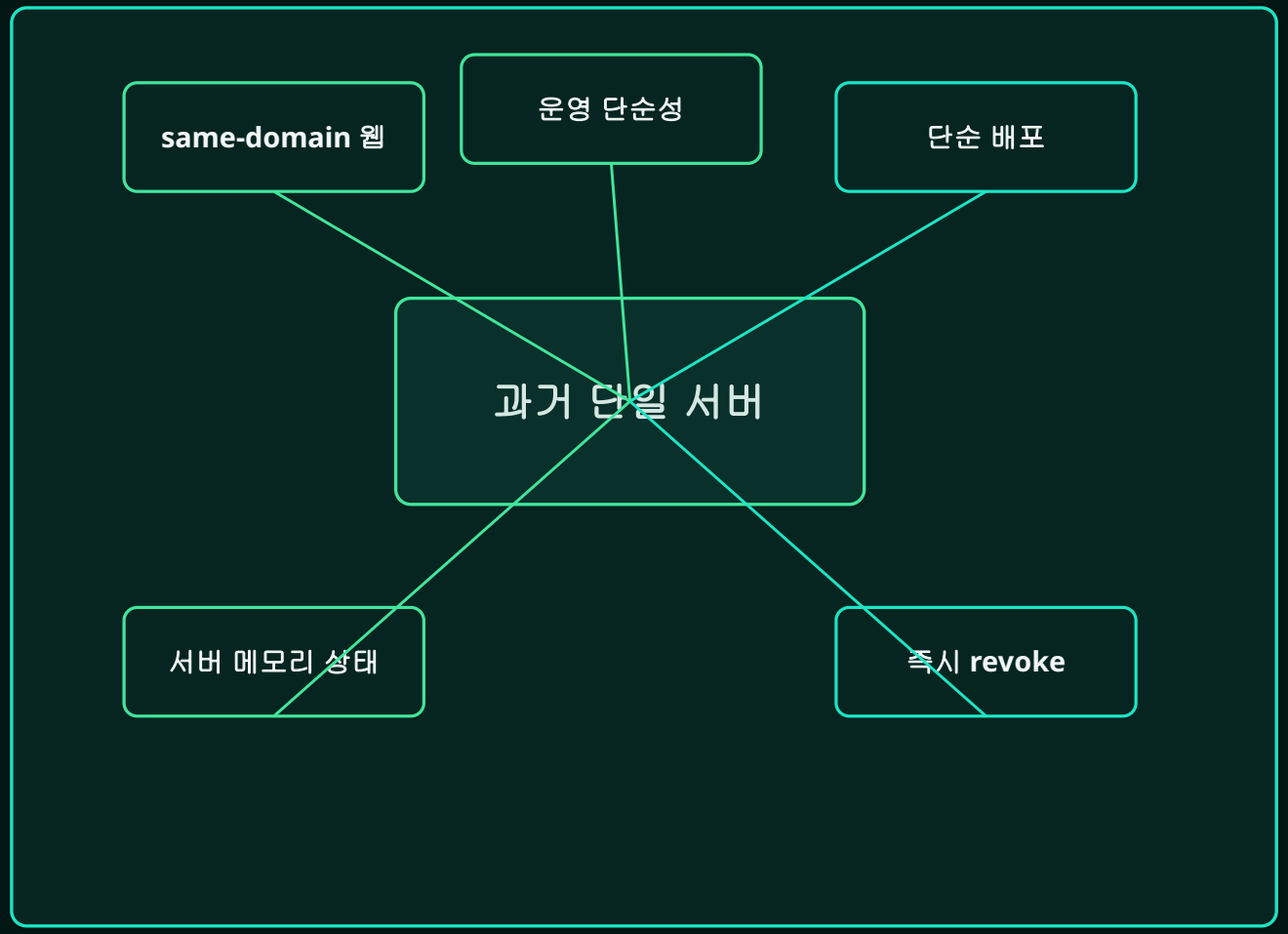
서버가 한 대 또는 소수였고, 상태를 로컬 메모리에 두어도 운영 가능했다

02

웹 클라이언트 중심이라 쿠키 세션이 UX와 맞았다

03

비밀번호·세션·권한을 한 코드베이스에서 다루는 것이 단순했다



단일 서버 세션의 장점

비밀번호와 세션

01

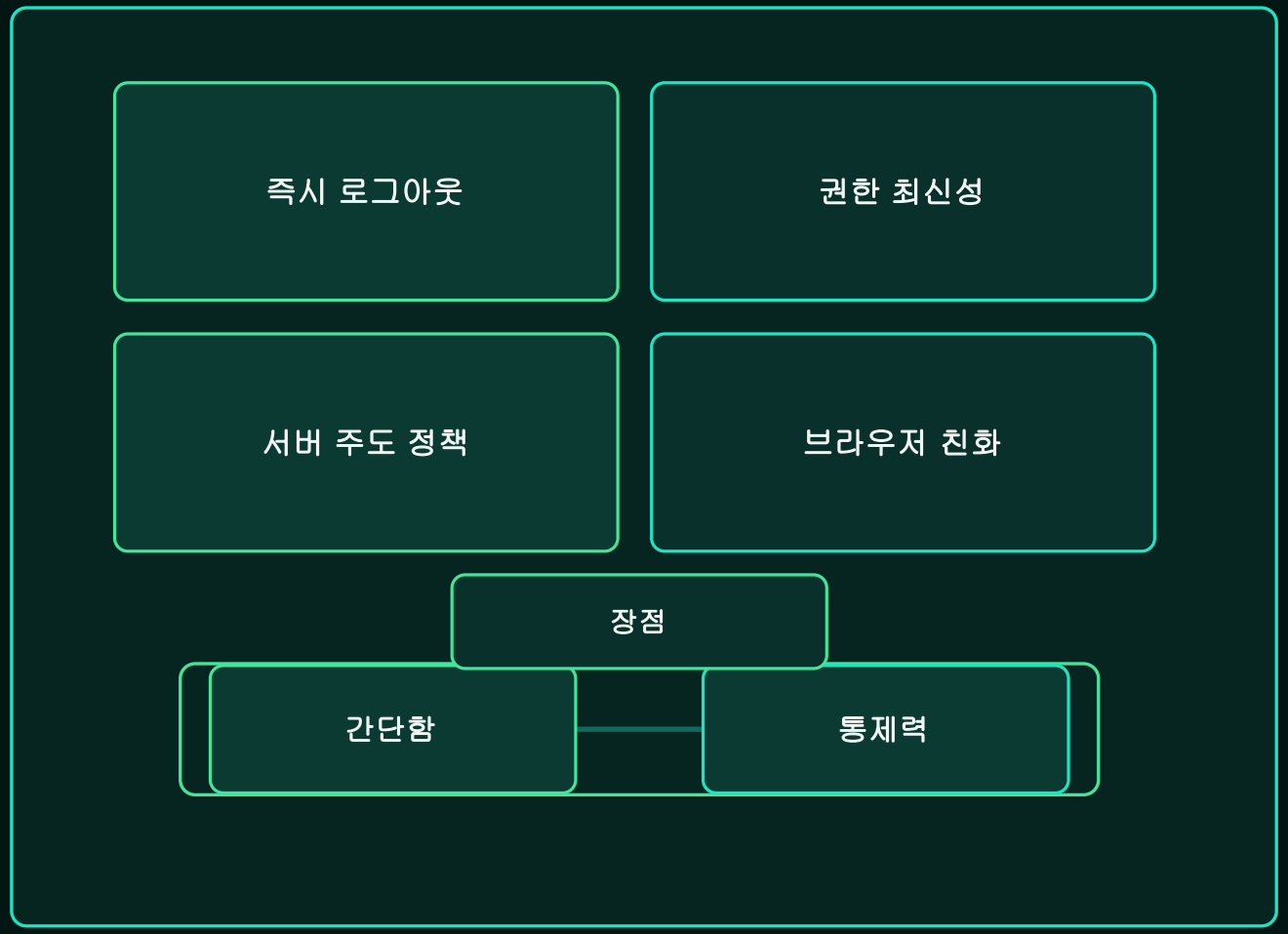
구현과 디버깅이 쉽고, 인증 상태를 한곳에서 통제하기 쉽다

02

즉시 강제 로그아웃·계정 잠금·권한 변경 반영이 자연스럽다

03

브라우저 기반 사내 도구나 소규모 서비스에는 아직도 매우 적합하다



단일 서버 세션의 한계

비밀번호와 세션

01

수평 확장 시 같은 사용자가 항상 같은 서버로 가야 하거나 세션이 끊긴다

02

인스턴스 재시작·배포·장애 시 로그인 상태가 사라질 수 있다

03

모바일·외부 API·여러 도메인 환경으로 넓어질수록 단순성이 줄어든다

scale-out 어려움

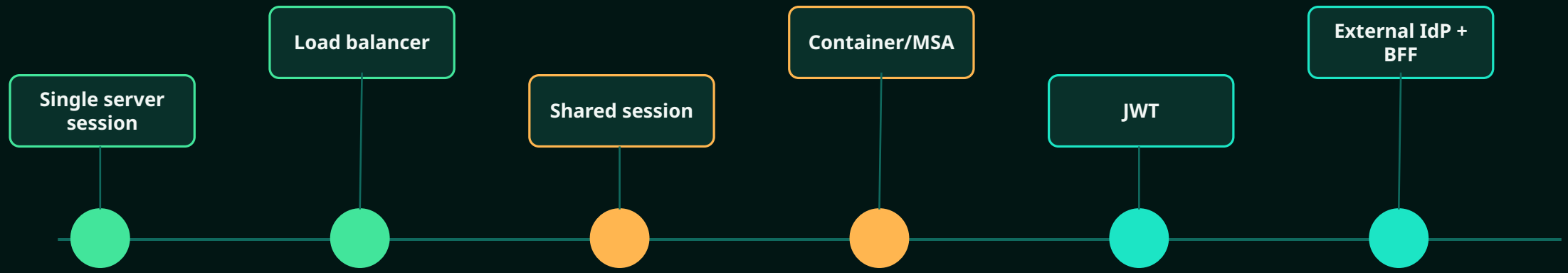
장애 시 세션 손실

서버 재배치 취약

모바일/API 확장 한계

진화 방향

좋은 기술의 등장이라기보다 상태 위치와 신뢰 경계의 이동



서버 상태 중심 → 진화의 방향 → 클라이언트 /외부 경계 중심

서버가 여러 대가 되면 첫 번째로 부딪히는 문제

01

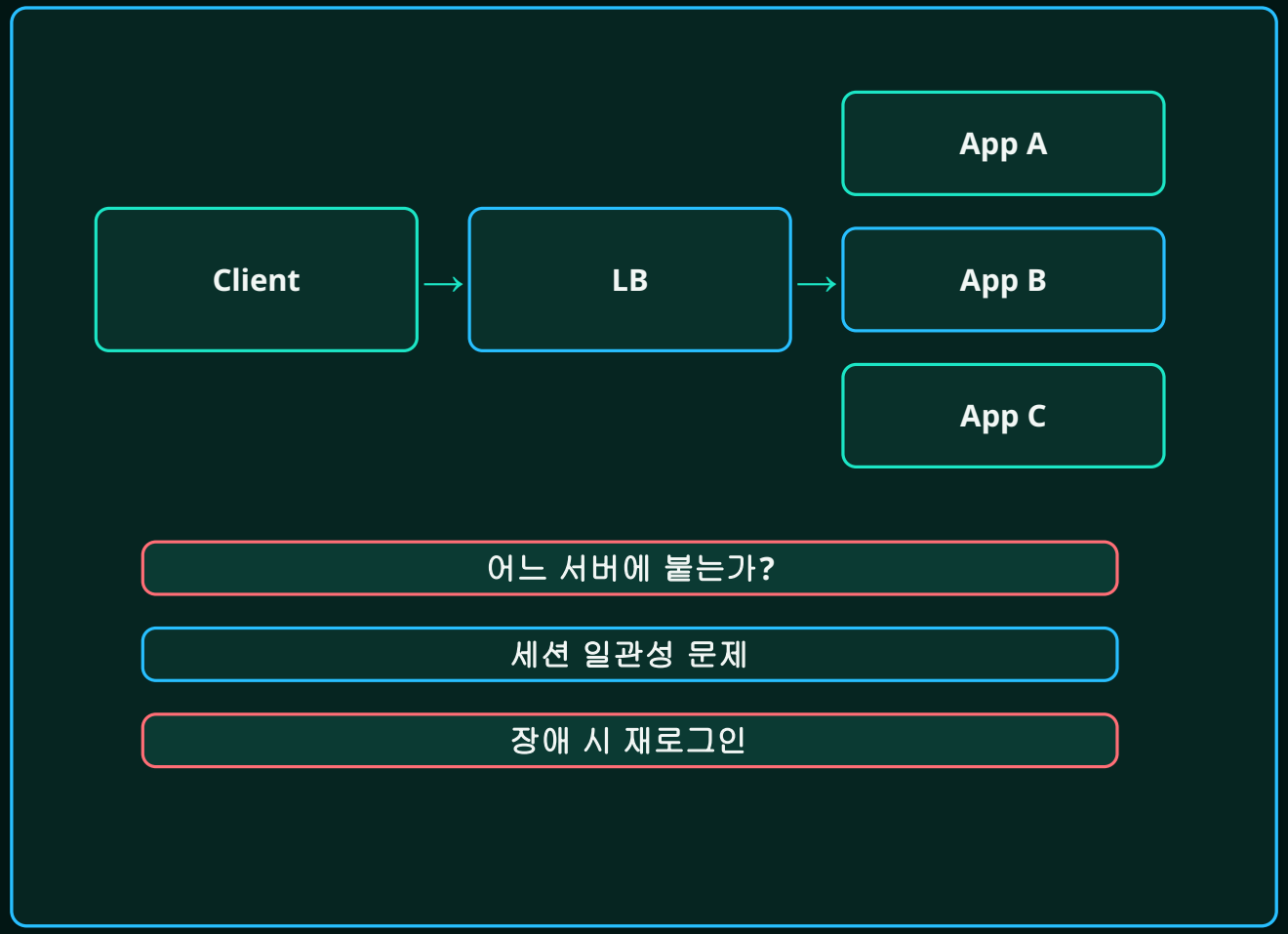
로드밸런서 뒤 여러 인스턴스가 뜨면 로컬 메모리 세션이 분산된다

02

A 서버에서 로그인한 사용자가 B 서버로 가면 상태를 못 읽는다

03

Sticky session 또는 중앙 저장소라는 선택지가 나온다



선택지 1: Sticky Session, 선택지 2: Shared Session

Sticky Session

01

로드밸런서가 사용자를 특정 서버에 붙임

02

구현은 간단하지만 장애·재배치에 약함

Shared Session

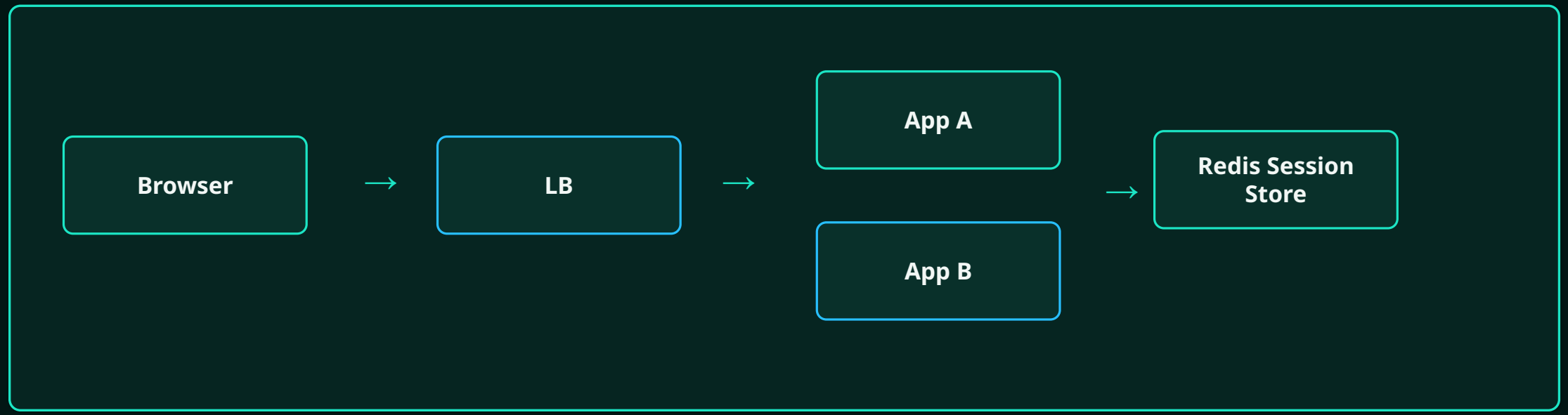
01

모든 앱이 같은 세션 저장소를 읽음

02

네트워크 홉이 늘지만 확장성이 나아짐

전형적인 Redis 기반 공유 세션 구조



공유 세션이 가져온 이점

01

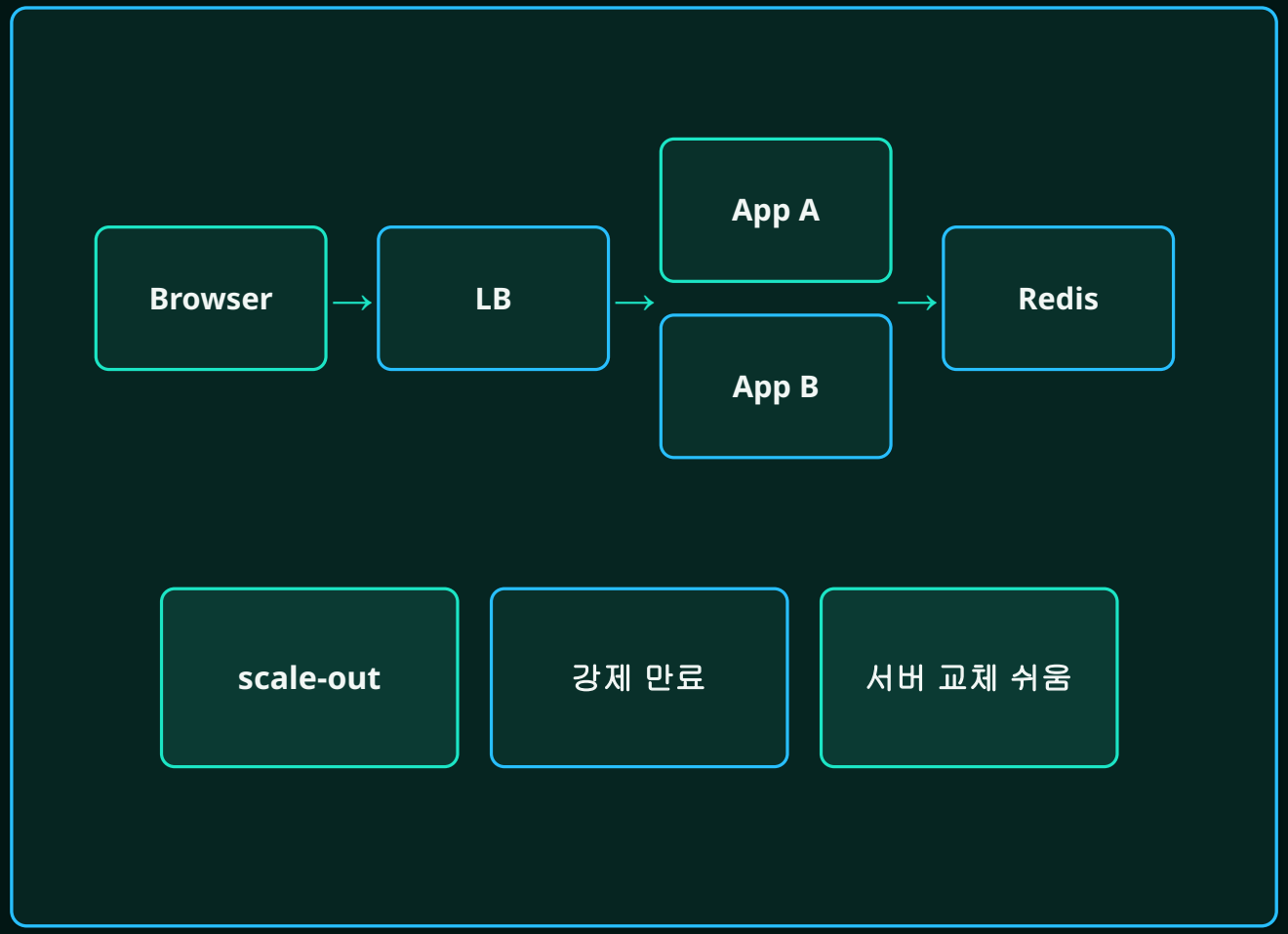
앱 인스턴스를 수평 확장해도 로그인 상태를 유지하기 쉽다

02

세션 삭제로 즉시 로그아웃·강제 만료를 계속 지원할 수 있다

03

브라우저 쿠키 기반 UX를 유지하면서 운영 유연성을 얻는다



하지만 비용도 함께 생긴다

01

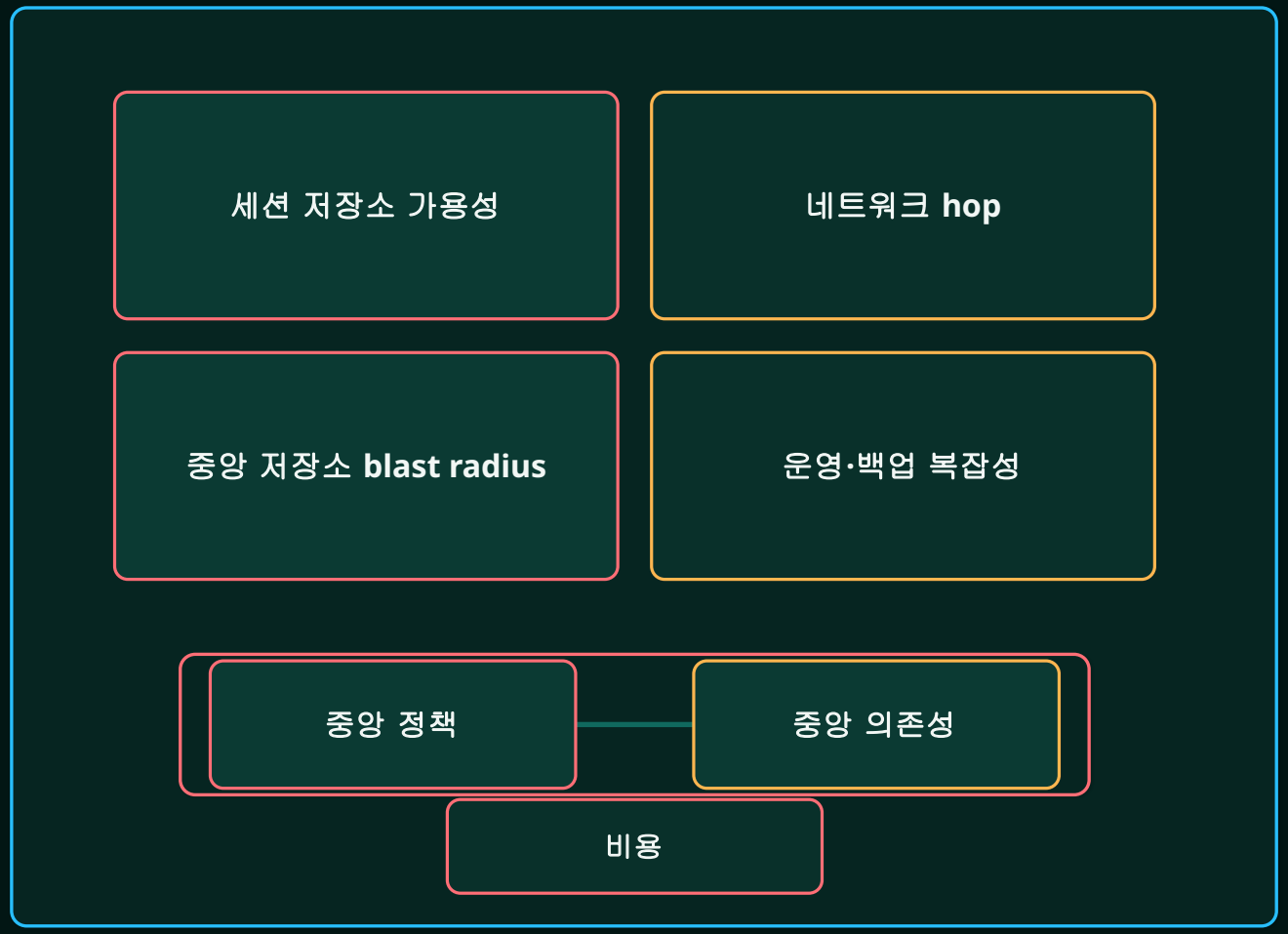
매 요청마다 외부 저장소 조회가 들어가 네트워크 지연이 증가한다

02

세션 저장소 장애가 곧 전체 로그인 장애가 된다

03

TTL, eviction, hot key, failover 같은 운영 문제가 보안 문제와 얽힌다



공유 세션의 보안 포인트

01

세션 ID는 충분히 예측 불가능해야 하고, 로그인 직후 회전해야 한다

02

저장소 접근 권한, 네트워크 분리, TLS, 백업 관리가 중요하다

03

세션에 과도한 개인정보를 넣지 말고 최소한의 권한 정보만 둔다

세션 ID 생성 강도

Redis 네트워크 분리

TTL·idle timeout

쿠키 Secure/HttpOnly/SameSite

회전, 타임아웃, 강제 만료는 세션의 강점이다

01

절대 만료와 유휴 만료를 분리해 설계한다

02

권한 상승, 비밀번호 변경, 관리자 차단 시 세션을 재발급하거나 폐기한다

03

다중 기기 세션 정책도 데이터 모델로 명시한다

멀티 리전과 재해 복구가 들어오면 다시 복잡해진다

01

지역 간 레이턴시와 복제 지연 때문에 세션 일관성 문제가 생긴다

02

리전 장애 시 세션 복구 정책이 곧 사용자 경험 정책이 된다

03

글로벌 서비스에서는 중앙 상태가 병목 또는 단일 실패점이 되기 쉽다

공유 세션이 잘 맞는 상황

01

브라우저 중심 서비스이고, 즉시 로그아웃과 권한 변경 반영이 중요하다

02

같은 조직 안에서 앱·도메인·배포를 통제할 수 있다

03

토큰 연동보다 운영 단순성이 더 중요한 내부 시스템이다

same-domain 웹 중심

즉시 revoke 중요

노드 수 예측 가능

모바일/API 비중 낮음

그런데 인프라가 바뀌었다: Container

Container와 MSA

01

컨테이너는 필요한 파일과 설정을 함께 묶은 격리된 프로세스 단위다

02

VM보다 가볍고 빠르게 뜨고 사라지며, 동일한 이미지를 여러 환경에 배포하기 쉽다

03

애플리케이션 인스턴스가 더 짧게 생겼다 사라지는 방향으로 운영 문화가 바뀌었다

기존 서버

01

상태가 서버에 오래 남음

02

배포 주기가 느림

Container 시대

01

인스턴스 교체 빠름

02

로컬 상태 가치 하락

컨테이너 vs VM

Container와 MSA

VM

01

게스트 OS 포함

02

격리 강하지만 무겁고 느림

03

상태가 오래 남기 쉬움

Container

01

호스트 커널 공유

02

빠르고 가볍고 복제 쉬움

03

인스턴스가 일시적이기 쉬움

컨테이너화가 인증 구조에 미친 영향

Container와 MSA

01

재시작·오토스케일링·롤링 배포가 흔해지며
인스턴스 로컬 상태가 취약해졌다

02

이미지 불변성을 선호할수록 상태는 외부 저장소로
밀려난다

03

‘서버는 잊어도 되는 존재’라는 사고방식이
Stateless를 더 매력적으로 보이게 했다

오케스트레이션 환경에서는 인스턴스가 영속 자산이 아니다

Container와 MSA

01

스케줄러는 인스턴스를 옮기고, 재시작하고, 증감시킨다

02

디버깅도 '어느 서버'보다 '어느 요청 흐름' 중심으로 바뀐다

03

상태를 들고 있는 서버보다, 재생성 가능한 서버가 운영상 유리해졌다

이미지

Pod

재스케줄

새 인스턴스

로컬 메모리 상태 비영속

세션/캐시 외부화 필요

MSA

Container와 MSA

01

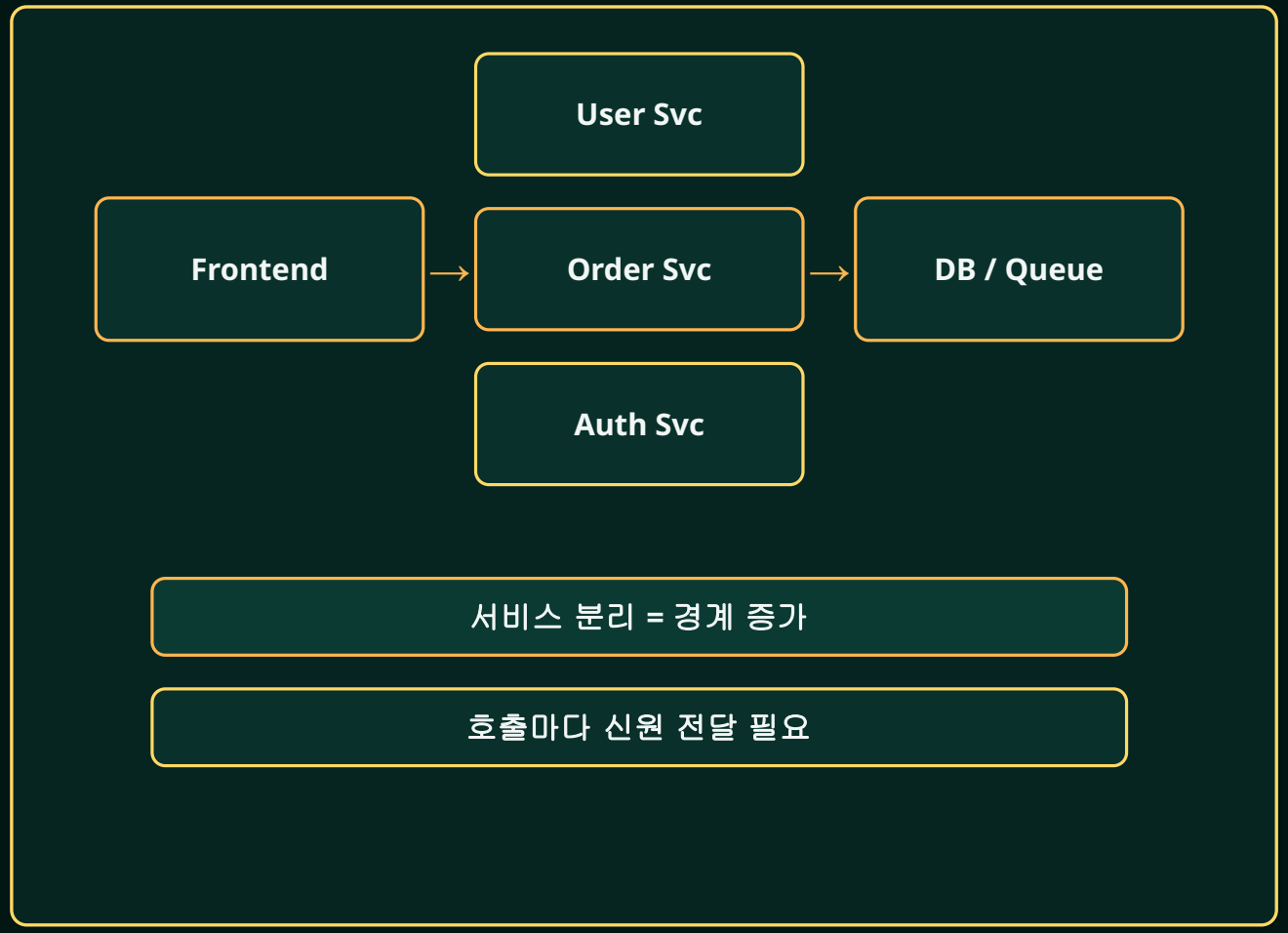
배포 단위, 데이터 소유권, 장애 전파, 조직 경계가 함께 바뀐다

02

인증 이후에 서비스 간 신뢰를 어떻게 전달할지 새 문제가 생긴다

03

‘한 앱의 세션’보다 ‘여러 서비스가 이해할 수 있는 주장(claim)’이 필요해진다



서비스 간 호출에서는 새로운 인증 문제가 생긴다

01

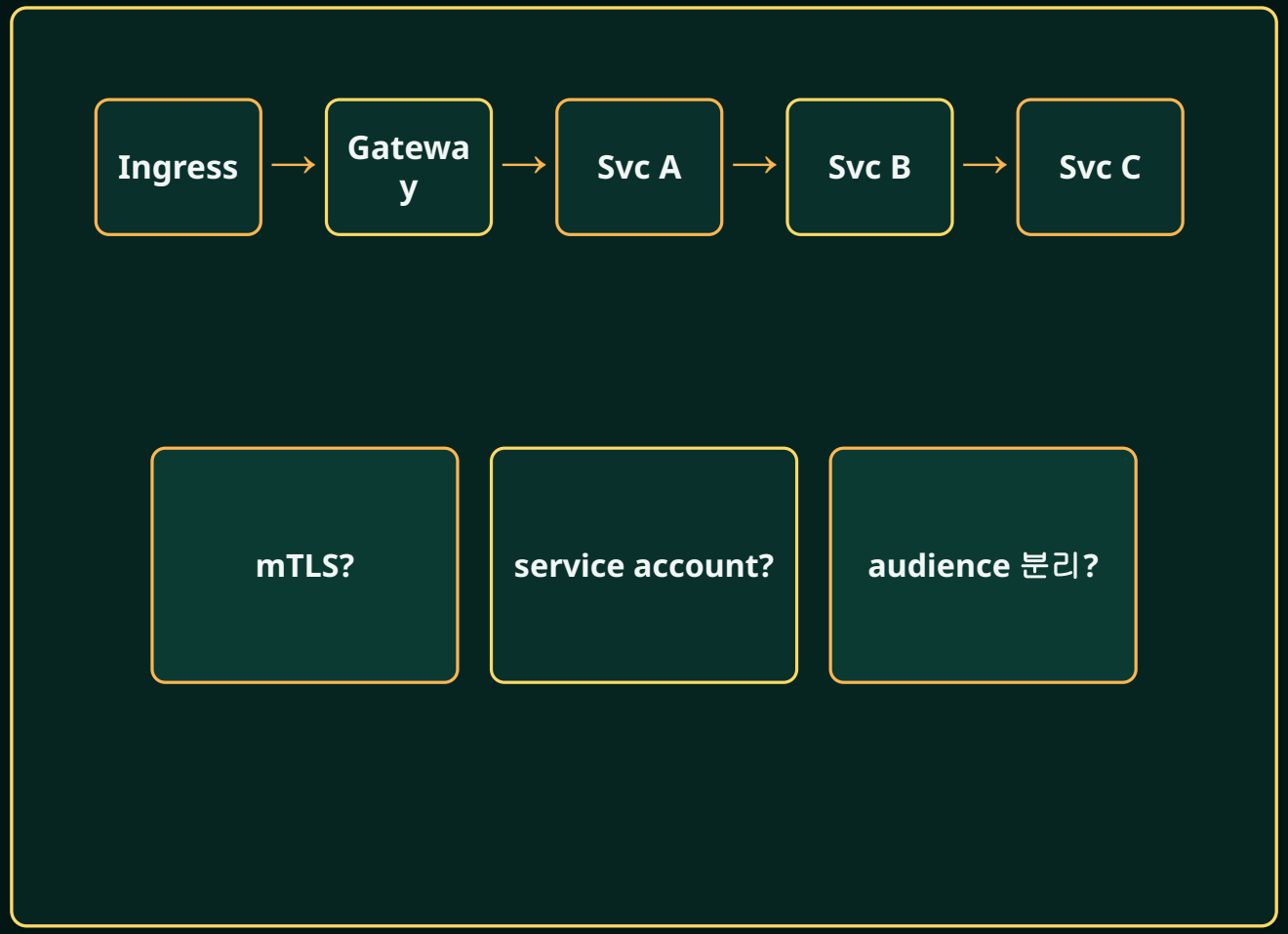
사용자 인증과 서비스 인증은 다르다

02

내부 API는 누가 호출했는지뿐 아니라 어떤 권한 범위로 호출했는지 알아야 한다

03

전달 토큰, mTLS, 서비스 계정, 정책 엔진 등의 조합이 등장한다



Gateway, Ingress, Service Mesh가 등장한 이유

Container와 MSA

01

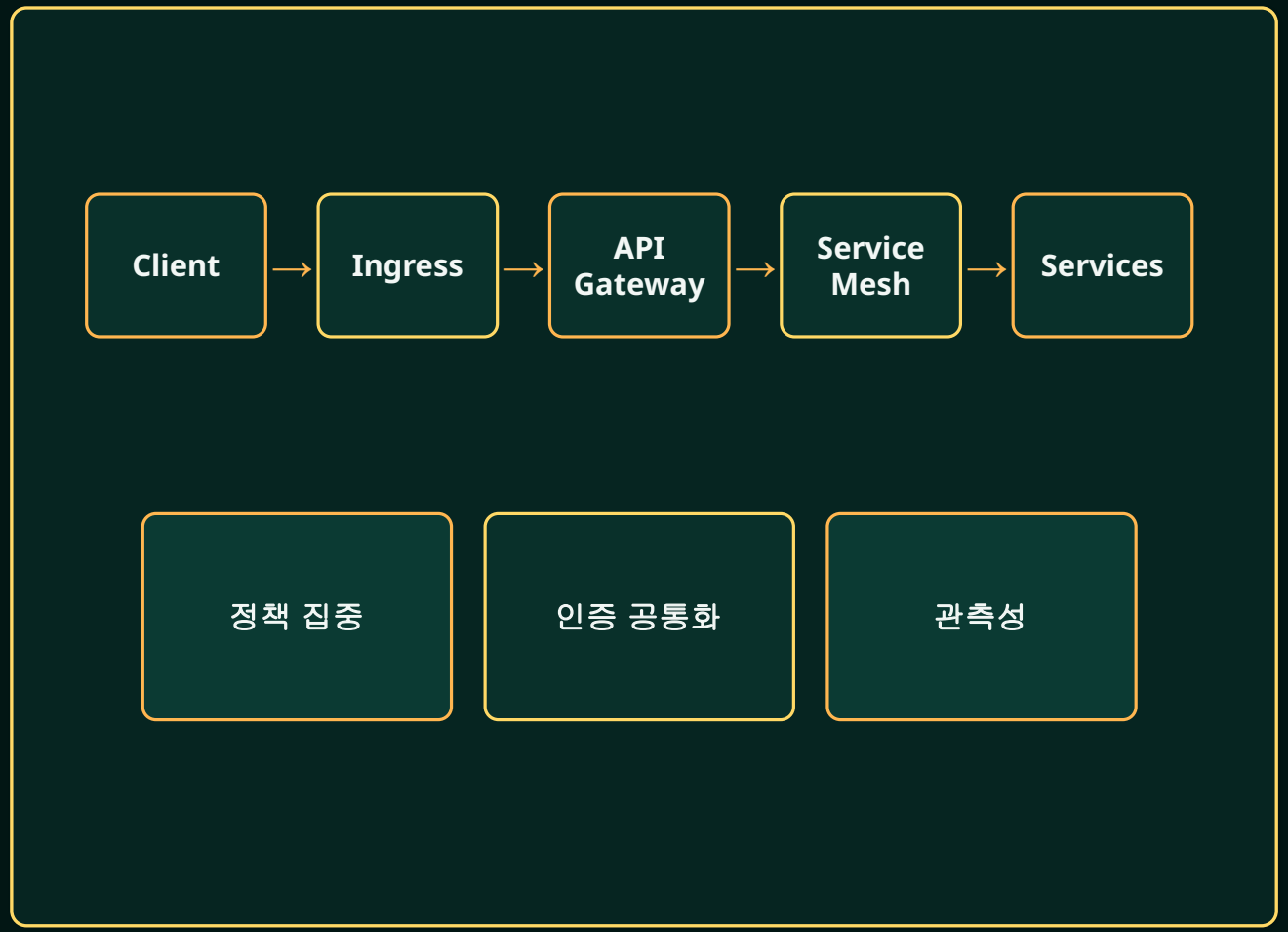
공통 정책(TLS, 인증, 라우팅, 제한)을 한곳에서 적용하려는 욕구

02

그러나 중앙화는 병목과 단일 실패점, 우회 경로라는 새 문제를 만든다

03

경계는 단순해지지만 내부 구조가 더 투명해지는 것은 아니다



왜 중앙 세션 저장소도 점점 부담이 되었나

Container와 MSA

01

모든 요청이 중앙 저장소를 거치면 레이턴시와 장애 영향이 커진다

02

서비스별 독립 배포와 팀 자율성이 커질수록 공유 상태 합의가 어려워진다

03

모바일·SPA·외부 파트너까지 연결되며 브라우저 쿠키 모델만으로 설명하기 어려워졌다

store latency

multi-region

failover

blast radius

이 시점에서 JWT가 매력적으로 보인다

01

자체 검증 가능한 토큰이면 서비스가 중앙 세션 조회 없이 신원을 해석할 수 있다

02

외부 IdP와의 연동, 마이크로서비스 간 전달, 모바일 클라이언트 지원이 쉬워 보인다

03

하지만 이 '쉬워 보임'이 이후의 함정을 함께 데려온다

플랫폼 변화가 인증 구조를 밀어 움직였다

Container와 MSA

01

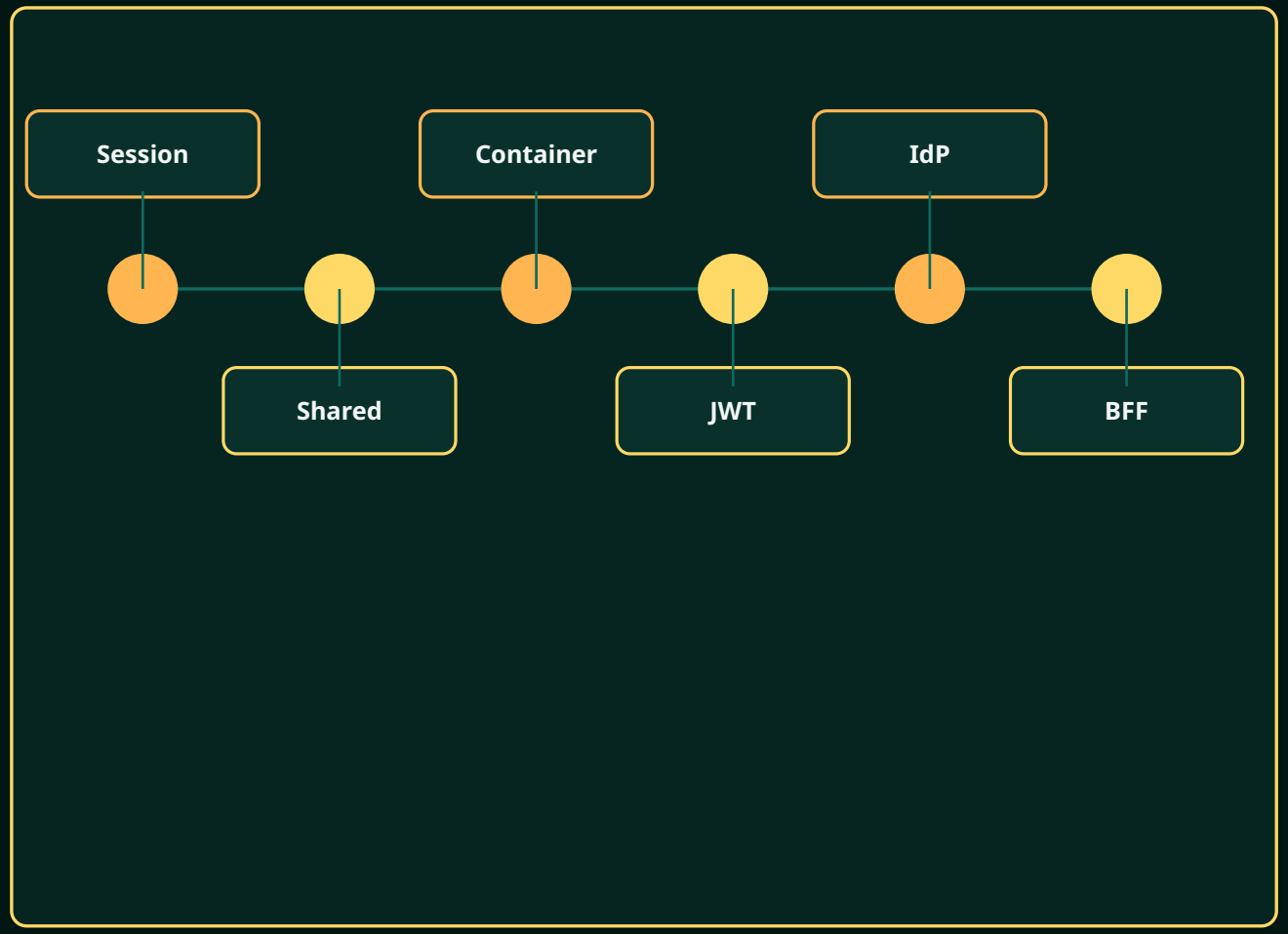
Single server → Shared session → Stateless token 은 운영 제약의 변화에 대한 응답이다

02

각 단계는 이전 문제를 줄였지만, 동시에 새 보안·운영 비용을 만들었다

03

그래서 '무엇이 최고인가'보다 '무엇을 포기할 수 있는가'가 질문이 된다



JWT는 무엇인가

JWT

01

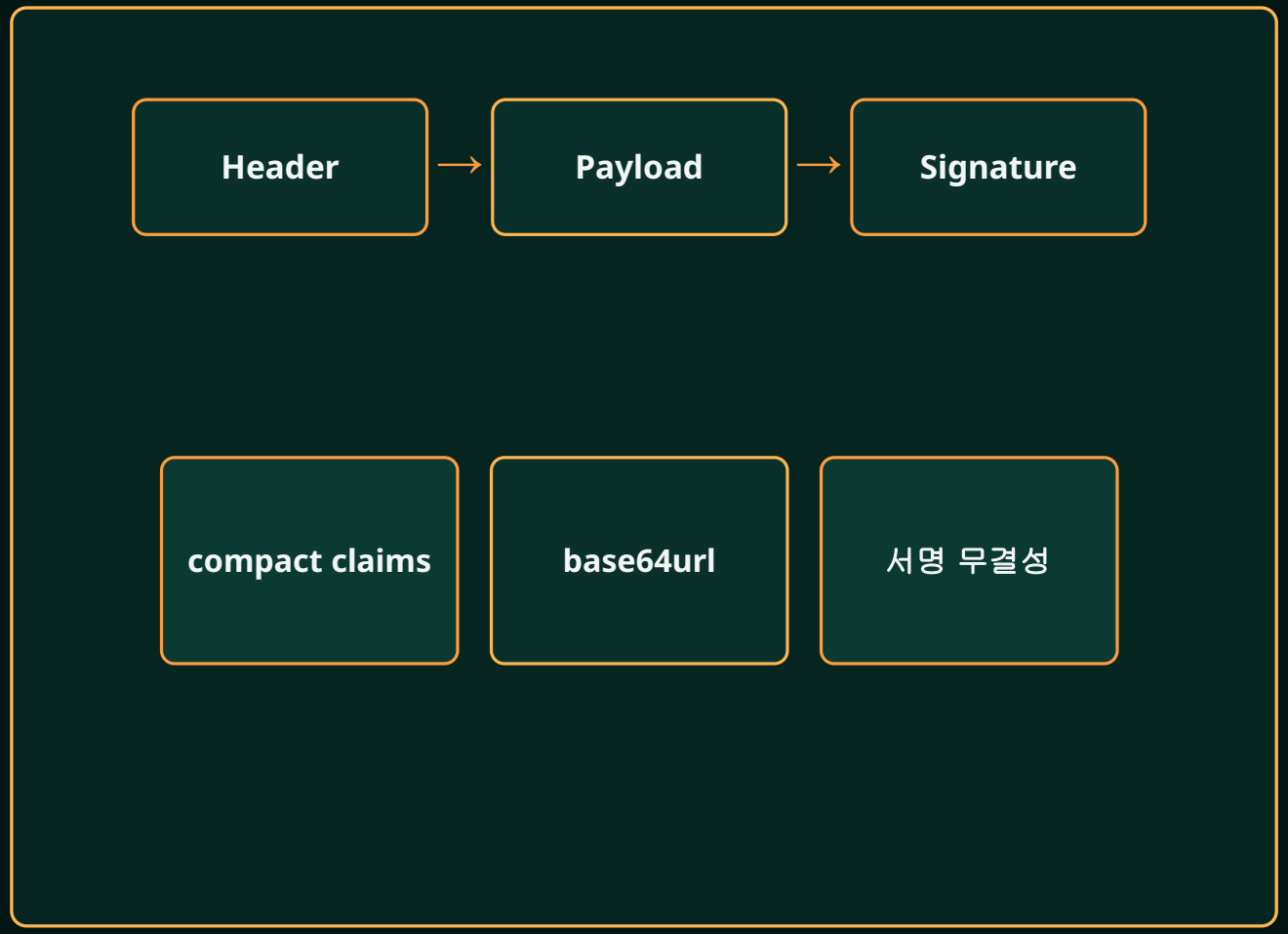
JWT는 claims를 표현하는 compact 토큰 형식이다

02

보통 Header.Payload.Signature 형태의 JWS로 많이 본다

03

토큰 형식일 뿐, 세션 정책·인가 모델·로그아웃 전략을 대신 설계해 주지 않는다



왜 팀들이 JWT를 좋아했나

JWT

01

중앙 세션 조회 없이 각 서비스가 토큰을 검증할 수 있다

02

모바일, SPA, 파트너 API까지 같은 형식을 공유하기 쉽다

03

외부 IdP·SSO와 연동할 때 표준 생태계가 풍부하다

stateless

서비스 간 전달

다양한 클라이언트

캐시 친화

JWT가 잘 해결하는 문제

JWT

01

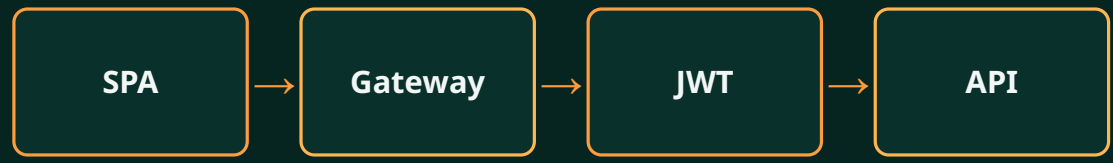
서버 간 상태 공유 비용을 줄여 수평 확장에 유리하다

02

발급자·대상·만료 같은 표준 claim으로 연동이 편해진다

03

API 보안과 연합 인증(federation)의 공통 언어로 쓰기 좋다



- 중앙 세션 의존 감소
- cross-domain에 적합
- public API 친화

하지만 JWT가 해결하지 못하는 것

JWT

01

즉시 로그아웃과 강제 회수는 여전히 별도 설계가 필요하다

02

권한 변경이 토큰 만료 전까지 반영되지 않을 수 있다

03

토큰을 어디에 저장하고 어떻게 재발급할지까지 해결하지는 않는다

logout 자체 해결 안 됨

권한 최신성 보장 안 됨

인가 모델 대체 불가

Claim는 많이 넣을수록 편하지만 위험도 커진다

JWT

01

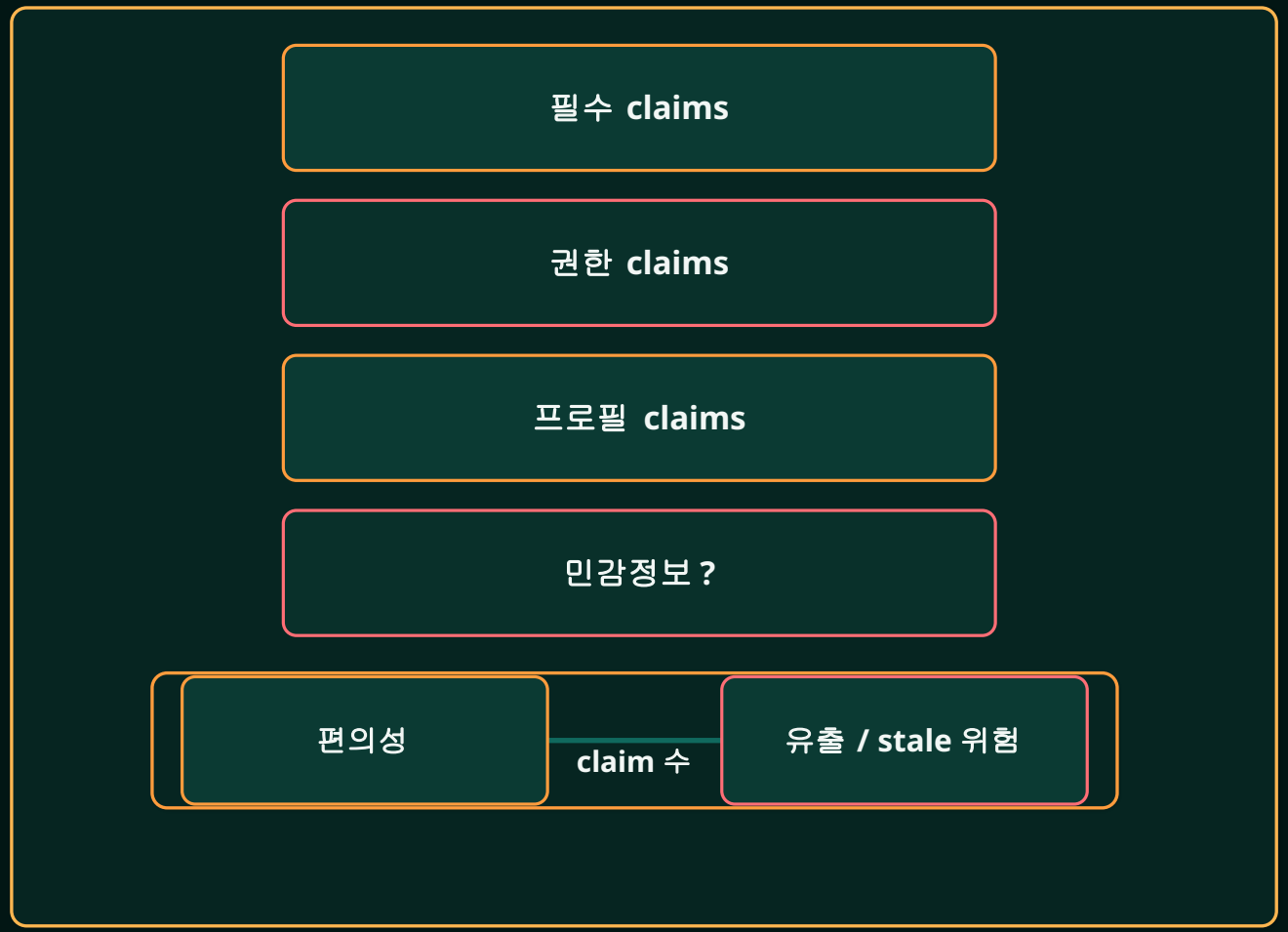
토큰이 커지면 네트워크 비용과 헤더 크기 문제가 생긴다

02

과도한 역할·조직·개인정보를 넣으면 노출 면적과 최신성 문제가 커진다

03

토큰에는 '검증에 꼭 필요한 최소 주장'만 넣는 편이 안전하다



가장 큰 대가: 회수(revocation)와 최신성(staleness)

JWT

01

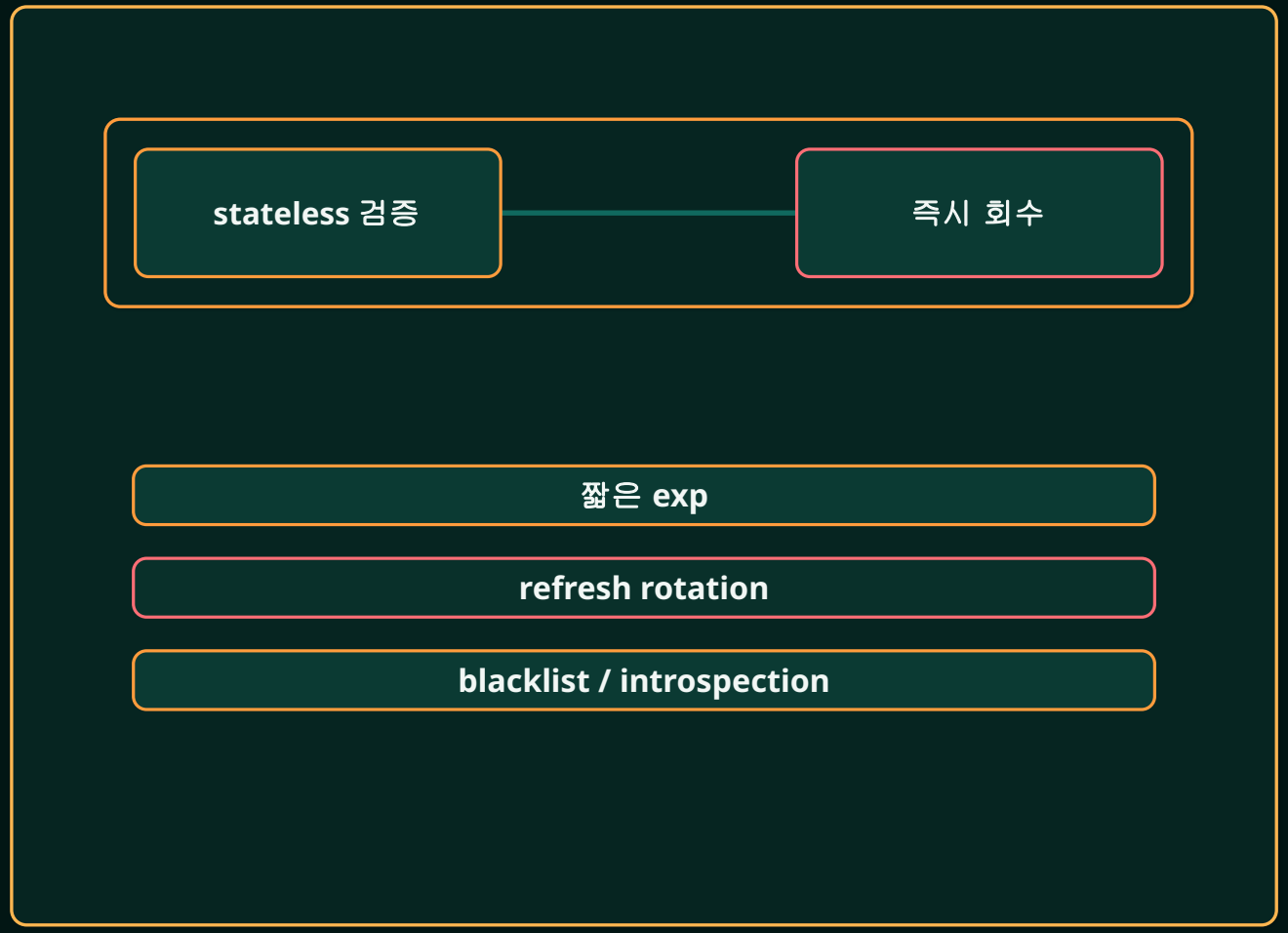
짧은 만료는 안전하지만 재발급 빈도와 UX 비용을 높인다

02

긴 만료는 운영은 편하지만 탈취 피해와 권한 변경 반영을 늦춘다

03

블랙리스트, 세션 버전, introspection, token exchange 같은 보완책이 나온다



키 관리가 보안의 중심으로 올라온다

JWT

01

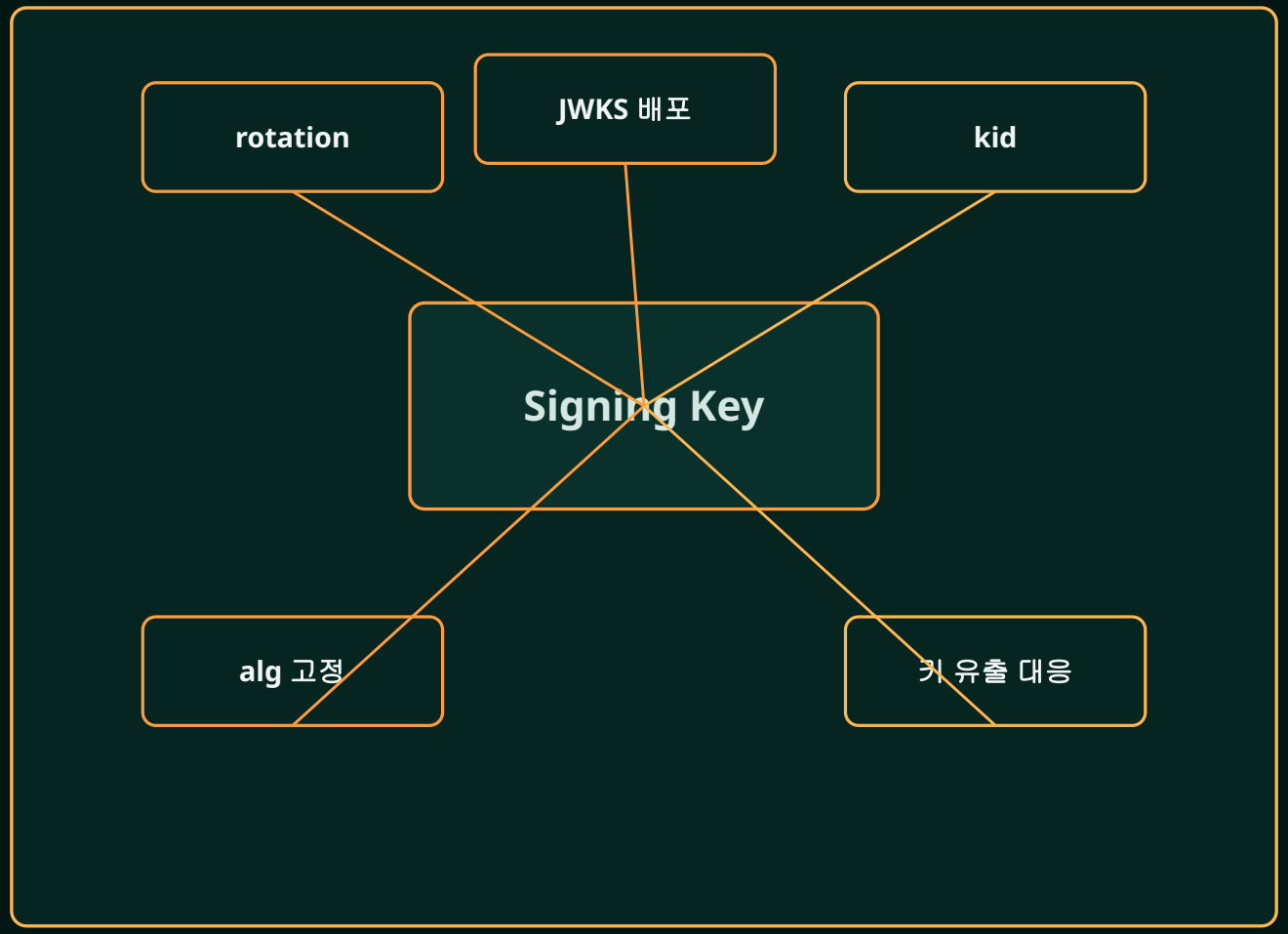
서명 키 유출은 토큰 발급권 유출과 같다

02

alg 검증, kid 처리, 키 회전, JWKS 캐시 정책을 잘못 잡으면 검증이 흔들린다

03

키는 코드·이미지·환경변수에 장기 보관하지 말고 별도 관리 체계를 둔다



JWT가 맞는 경우와 아닌 경우

JWT

01

여러 서비스·도메인·클라이언트가 표준 토큰을 공유해야 할 때는 강하다

02

반대로 same-site 웹앱, 즉시 로그아웃 중요, 단순 운영 선호라면 세션이 더 낫다

03

JWT는 '더 최신'이 아니라 '다른 비용 구조를 가진 선택지'다

SPA / mobile

public API

서비스 간 신원 전달

즉시 revoke는 약함

인증 주체가 애플리케이션 밖으로 나간 이유

External IdP

01

모든 서비스가 비밀번호·MFA·계정 복구를 직접 구현하는 비용이 너무 컸다

02

조직 간 SSO, 소셜 로그인, 규제 준수 요구가 커졌다

03

애플리케이션은 인증 시스템이 아니라 비즈니스 시스템이 되기를 원했다

OAuth2와 OIDC의 차이

External IdP

01

OAuth2는 '권한 위임' 프레임워크다

02

OIDC는 OAuth2 위에 '사용자 인증과 프로필 전달'을 덧댄다

03

로그인에 OAuth2만 말하는 경우가 많지만, 실제 사용자 인증은 대개 OIDC 맥락이다

OAuth2

01

권한 위임

02

API 접근

OIDC

01

사용자 로그인

02

ID 토큰

외부 IdP의 장점

External IdP

01

SSO, MFA, 계정 잠금, 위험 기반 인증을 중앙에서 통합할 수 있다

02

애플리케이션은 비밀번호 저장과 복구 책임을 줄일 수 있다

03

조직 규모가 커질수록 보안 정책과 감사 일관성이 좋아진다

SSO

MFA

계정 라이프사이클

규정 준수

감사 통합

외부 IdP의 비용

External IdP

01

프로토콜과 운영 복잡도가 증가하고, 장애 영역이 외부로 확장된다

02

벤더 의존성, 가격, 커스터마이징 한계가 생긴다

03

토큰 수명, 클레임 설계, 계정 연결 정책을 앱이 여전히 고민해야 한다

- 외부 장애 의존
- 토큰 검증 복잡성
- 계정 연결 난이도
- logout mismatch

언제 외부 IdP가 가치 있는가

External IdP

01

여러 애플리케이션, 여러 조직, 여러 인증 수단을 통합해야 할 때

02

MFA·감사·계정 수명주기 관리가 중요한 엔터프라이즈 환경일 때

03

단일 앱·단일 도메인·단순 사용자 모델에서는 과투자일 수 있다

다수 제품 공통 로그인

enterprise federation

강한 계정 정책

전담 IAM 운영 가능

같은 API를 모든 클라이언트에 그대로 주면 왜 힘든가

BFF와 Edge

01

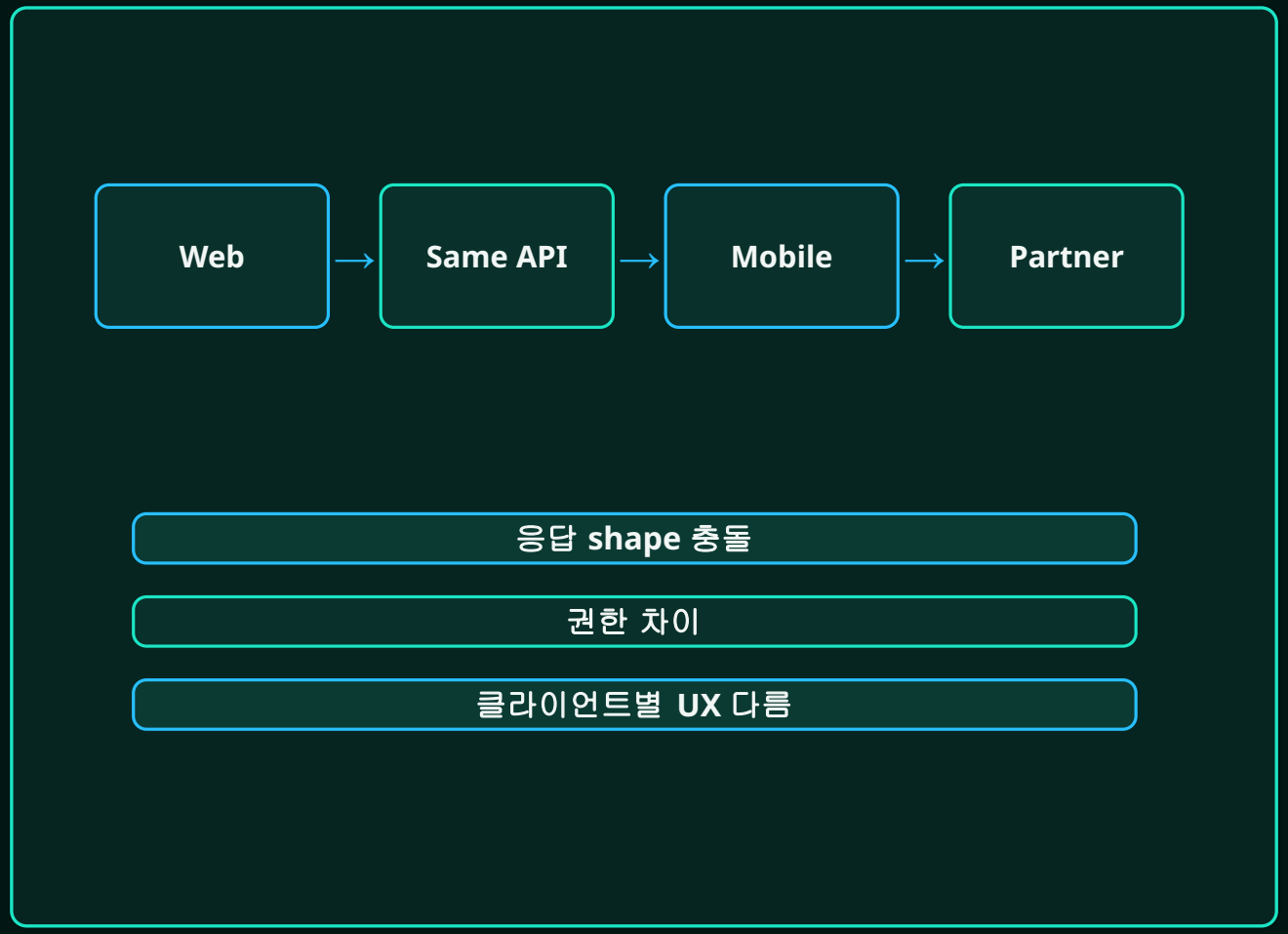
웹, 모바일, 파트너, 관리자 도구는 필요한 데이터와 보안 가정이 다르다

02

하나의 범용 API는 응답 과다, 권한 혼재, 버전 충돌을 부르기 쉽다

03

클라이언트 차이를 서버 경계에 반영하려는 시도로 BFF가 나온다



BFF란 무엇인가

BFF와 Edge

01

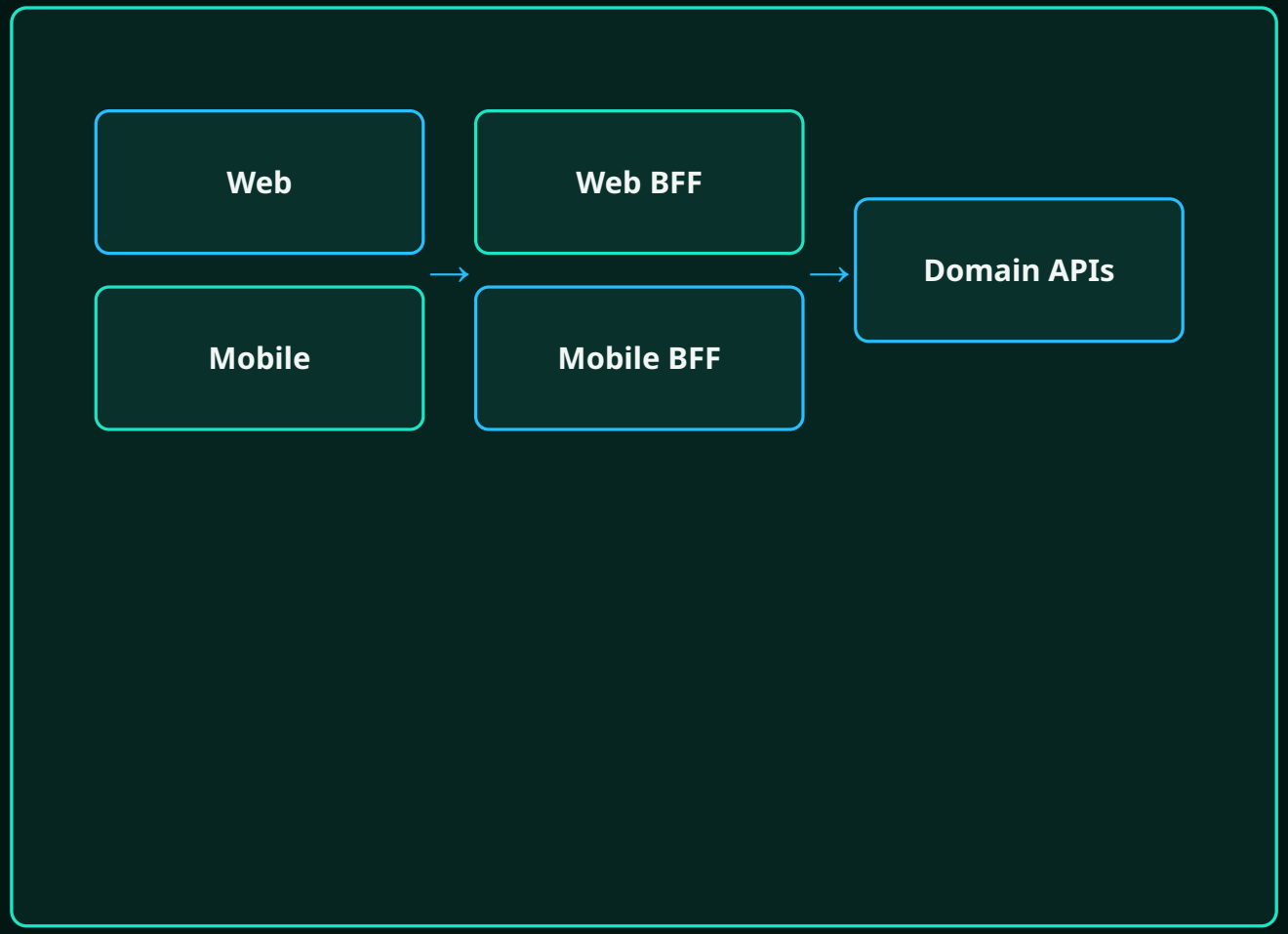
특정 프런트엔드에 맞춘 얇은 백엔드를 별도로 둔다

02

웹용, 모바일용, 관리자용이 서로 다른 응답 모델과 정책을 가질 수 있다

03

도메인 서비스는 공통으로 재사용하고,
표현·조합·보안 경계는 분리한다



BFF가 보안에 주는 이점

BFF와 Edge

01

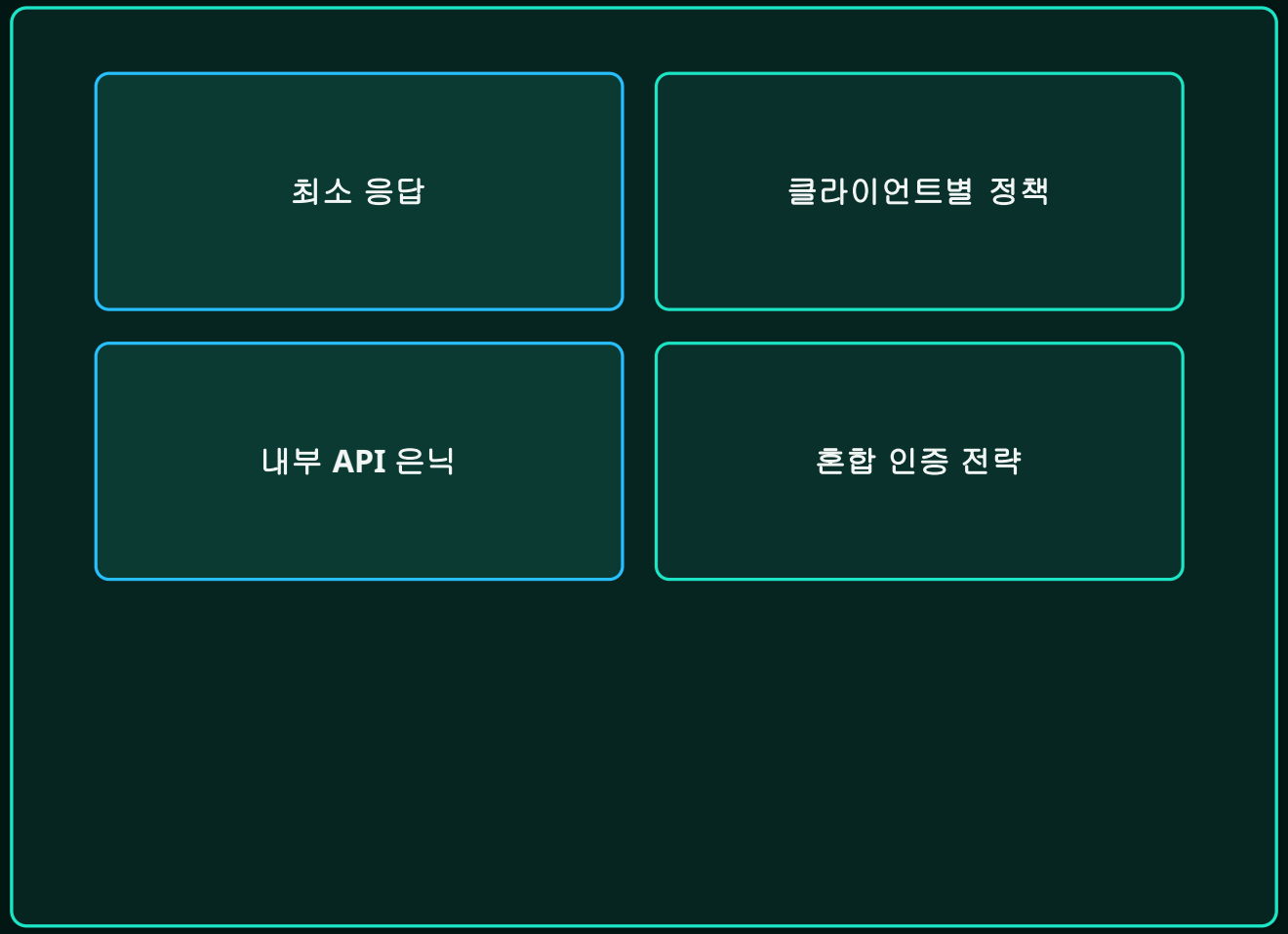
클라이언트별 최소 권한·최소 응답을 강제하기 쉬워진다

02

브라우저에는 cookie session, 내부 호출에는 token 같은 혼합 전략이 가능하다

03

비공개 API를 클라이언트로 직접 노출하지 않아 공격 표면을 줄일 수 있다



하지만 BFF도 공짜가 아니다

BFF와 Edge

01

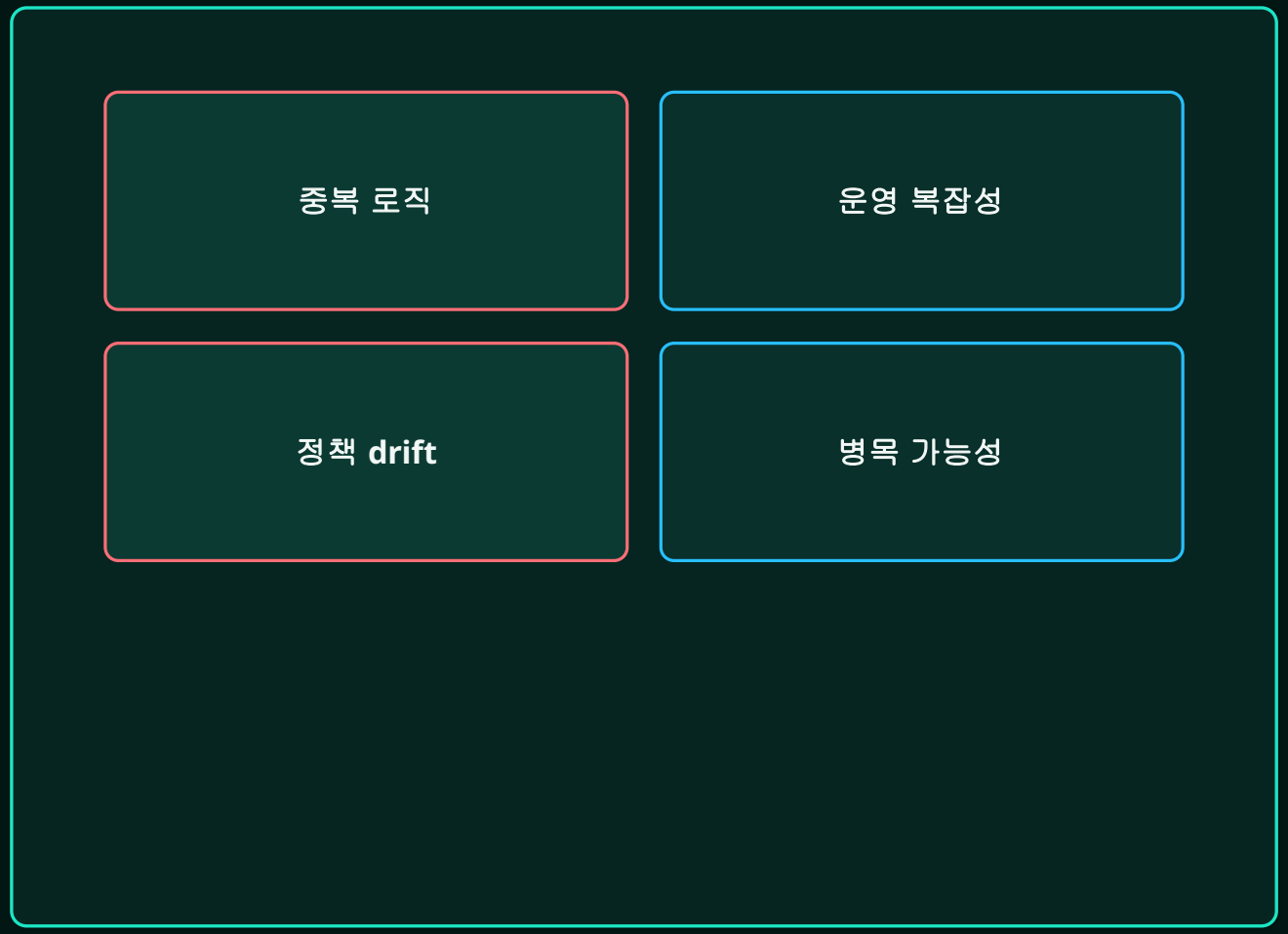
서비스 수와 배포 파이프라인이 늘어 운영 복잡도가 증가한다

02

로직 중복, 팀 경계 충돌, 캐시/세션 일관성 문제가 생길 수 있다

03

BFF가 두꺼워지면 또 다른 모놀리스 또는 우회 경로가 된다



웹, SPA, 모바일은 신뢰 표면이 다르다

BFF와 Edge

01

웹 same-site 앱은 쿠키 전략이 강하고, pure SPA는 토큰 저장 전략이 더 까다롭다

02

모바일은 브라우저와 다른 저장소·앱 간 전환 문제를 가진다

03

그래서 인증 패턴도 클라이언트별로 달라지는 것이 자연스럽다

웹: same-site cookie

SPA: XSS 주의

모바일: secure storage

파트너: token / scope

점진적 전환은 가능하다

BFF와 Edge

01

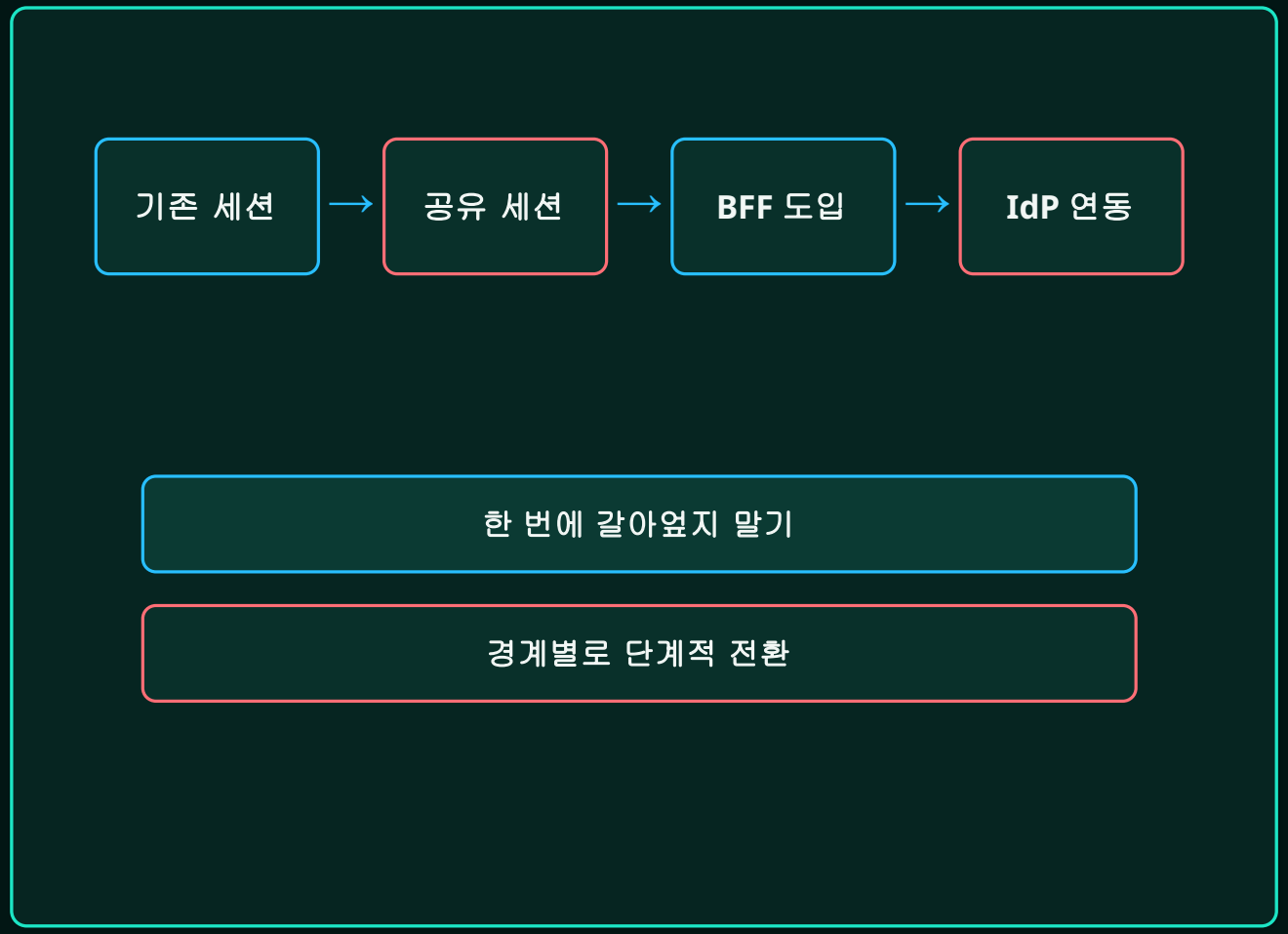
모놀리스를 한 번에 갈아엎기보다 로그인 경계부터 바꾼다

02

먼저 응답 DTO, 인가, 감사 로깅을 분리하고 나서 BFF를 추가해도 된다

03

아키텍처 이행은 기능 이전보다 '신뢰 경계 이전' 순서로 보는 편이 안전하다



실제 사고의 다수는 인증 이후에 난다

01

로그인 성공은 시작일 뿐이고, 그 뒤의 인가·정보노출·자원통제가 더 중요하다

02

관리자 API, 검색 API, 다운로드 API는 모두 권한이 달라질 수 있다

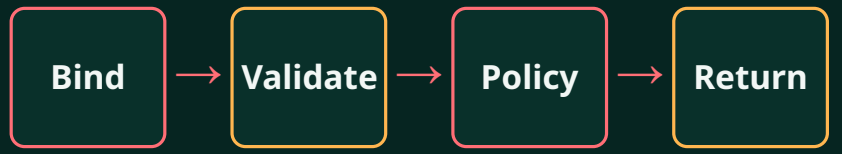
03

‘누가 들어왔나’보다 ‘무엇을 하게 할 건가’가 핵심이다

BOLA/IDOR: ID를 믿는 순간 생기는 문제


```
// 위험: ID만으로 조회
db.First(&post, postID)

// 안전: 소유권까지 함께 조회
db.Where("id = ? AND author_id = ?", postID,
actor.ID).First(&post)
```



객체 식별자 접근이 있는 모든

숫자 **ID**를 **UUID**로 바꿔도 인가를

필드 수준 인가가 필요한 순간

인가와 운영 보안

01

같은 사용자 객체라도 일반 사용자와 관리자가 볼 수 있는 필드는 다르다

02

응답 필터링과 수정 가능 필드 목록을 분리해 관리한다

03

‘한 번 조회했으니 다 보여 줘도 된다’는 가정이 정보 노출을 만든다

요청 객체 → 인가 규칙 → 응답 DTO

필드	응답 처리
id	노출
name	노출
role	상황별
salary	관리자만
private_note	내부 전용

Mass Assignment: 바인딩 가능한 필드가 너무 많다

01

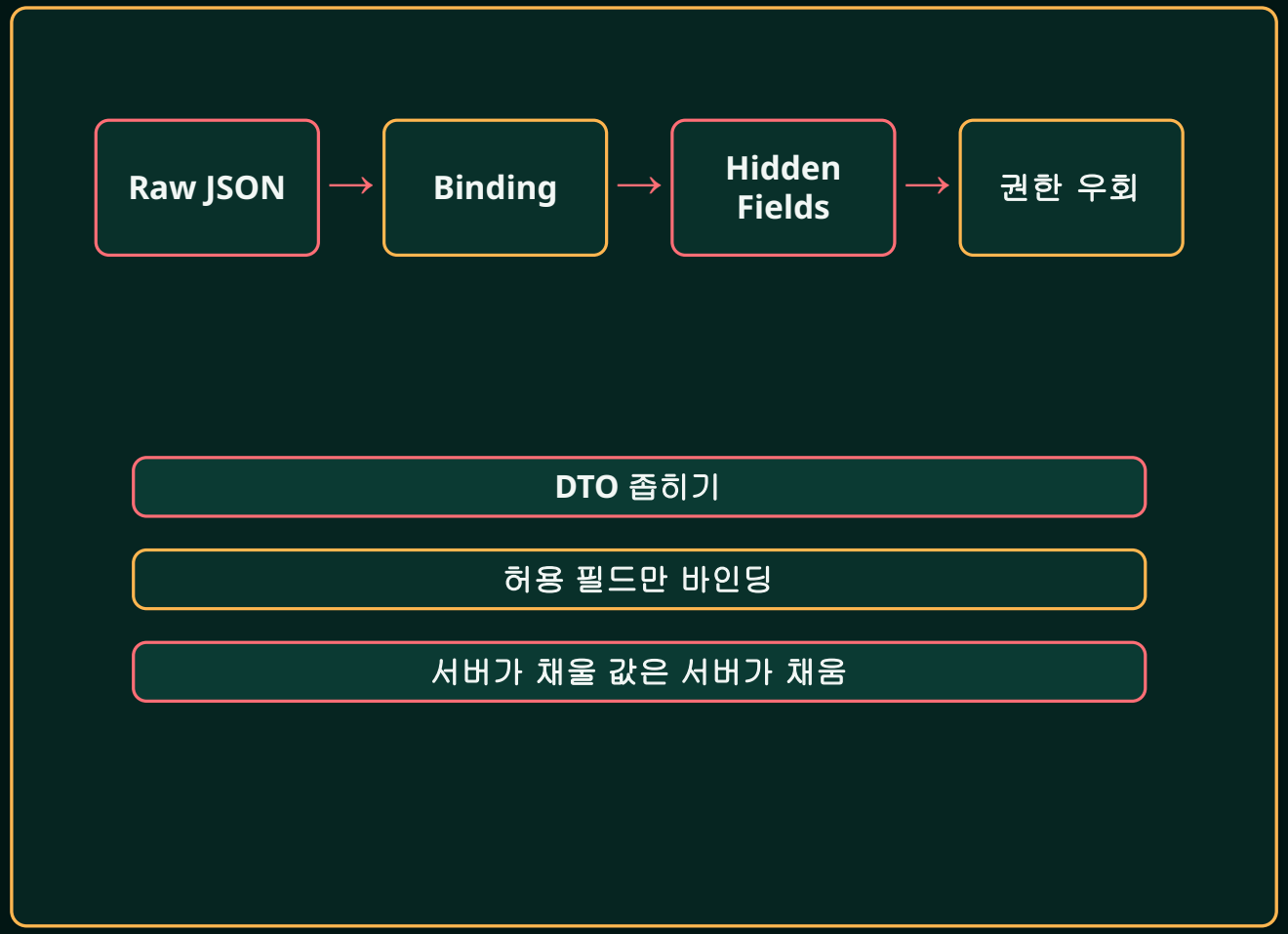
클라이언트가 보내면 안 되는 role, status, owner_id 같은 필드가 섞일 수 있다

02

입력 DTO와 도메인 모델을 분리해 쓰기 가능한 필드를 제한한다

03

편의성 때문에 model 전체 바인딩을 허용하면 사고가 난다



검색·필터·정렬·페이지네이션도 공격 표면이다

01

정렬 키, 비교 연산자, include 필드, 페이지 크기는 모두 비용과 노출을 키운다

02

허용 목록 기반의 정렬/필터 스키마가 필요하다

03

큰 page size, deep pagination, broad export는 DoS와 정보 수집에 악용된다

과도한 정보 노출을 줄이는 응답 설계

01

목록 응답은 최소 필드만, 상세 응답은 권한별 확장 필드만

02

내부 상태, soft-delete 플래그, 감사 메타데이터를 무심코 내보내지 않는다

03

응답 모델 버전과 가시성 정책을 명시적으로 관리한다

필드	응답 처리
profile	public DTO
private_email	owner/admin only
role	최소 노출
internal_flags	비노출

Rate limit, quota, cost control은 기능별로 설계한다

01

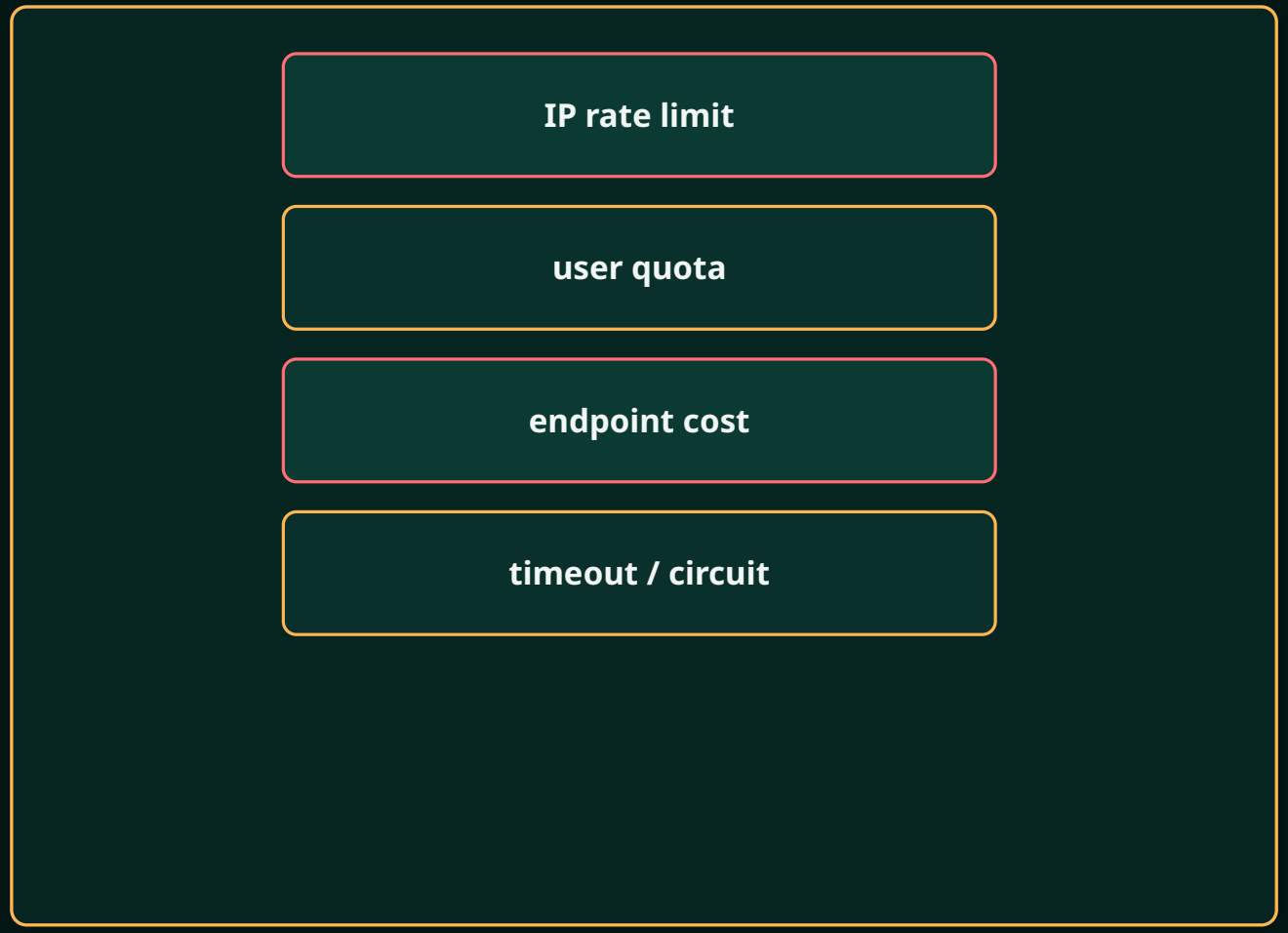
IP만이 아니라 사용자, 테넌트, API 키, 작업 종류 단위 제한이 필요하다

02

로그인, SMS, 이메일, 내보내기, 검색은 서로 다른 비용 모델을 가진다

03

거절(429) 정책과 사용자 메시지도 제품 설계의 일부다



DDoS보다 더 자주 만나는 것: 애플리케이션 레벨 자원 고갈

01

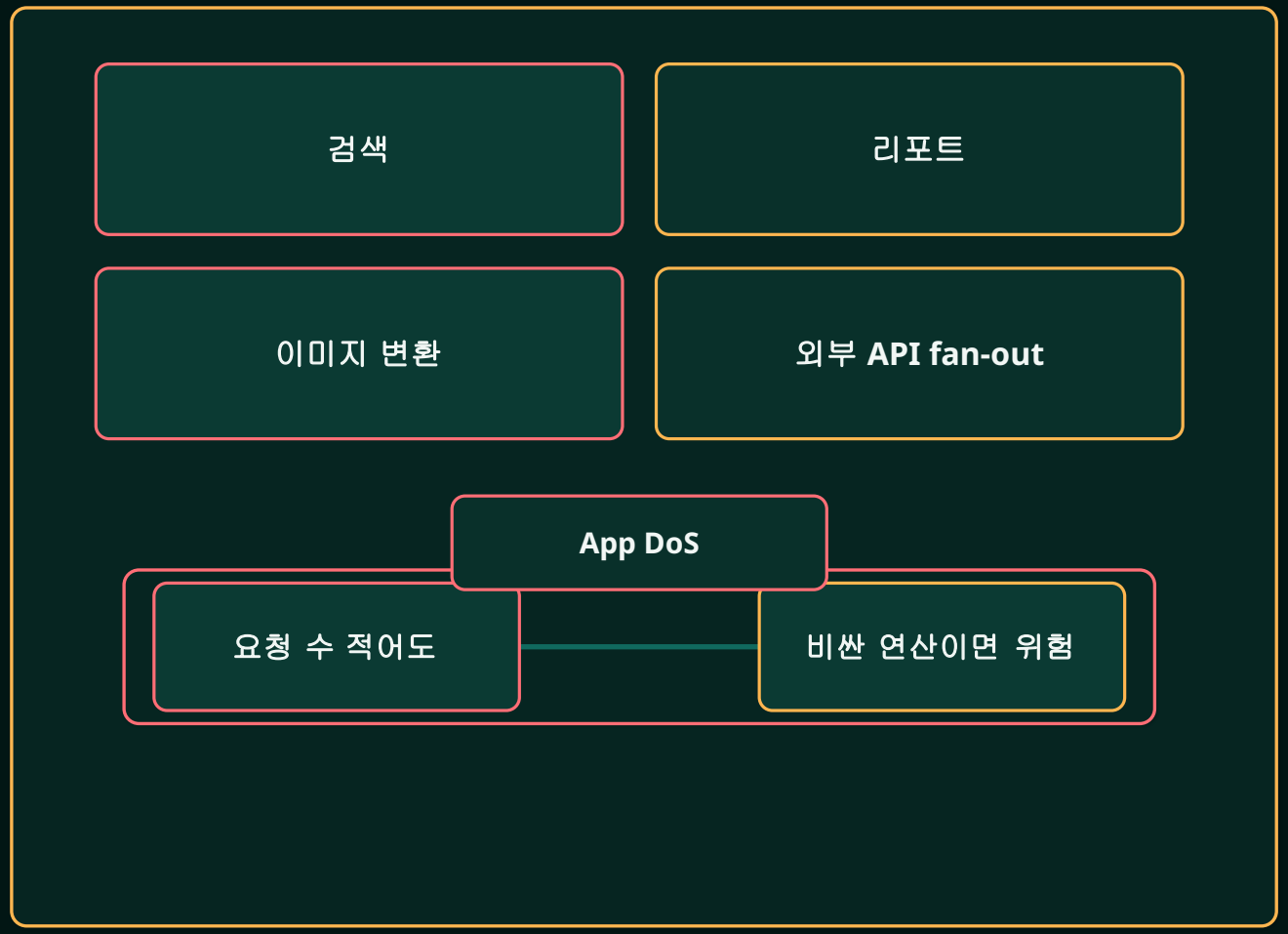
비싼 DB 쿼리, N+1, 외부 API fan-out, 큰 파일 생성이 공격 벡터가 된다

02

비동기 처리, 큐, 백프레셔, 캐시, circuit breaker로 비용 전파를 줄인다

03

정상 사용자의 폭주와 악의적 요청을 모두 견디는 설계가 필요하다



비밀 관리: 키, 토큰, 비밀번호, 인증서

인가와 운영 보안

01

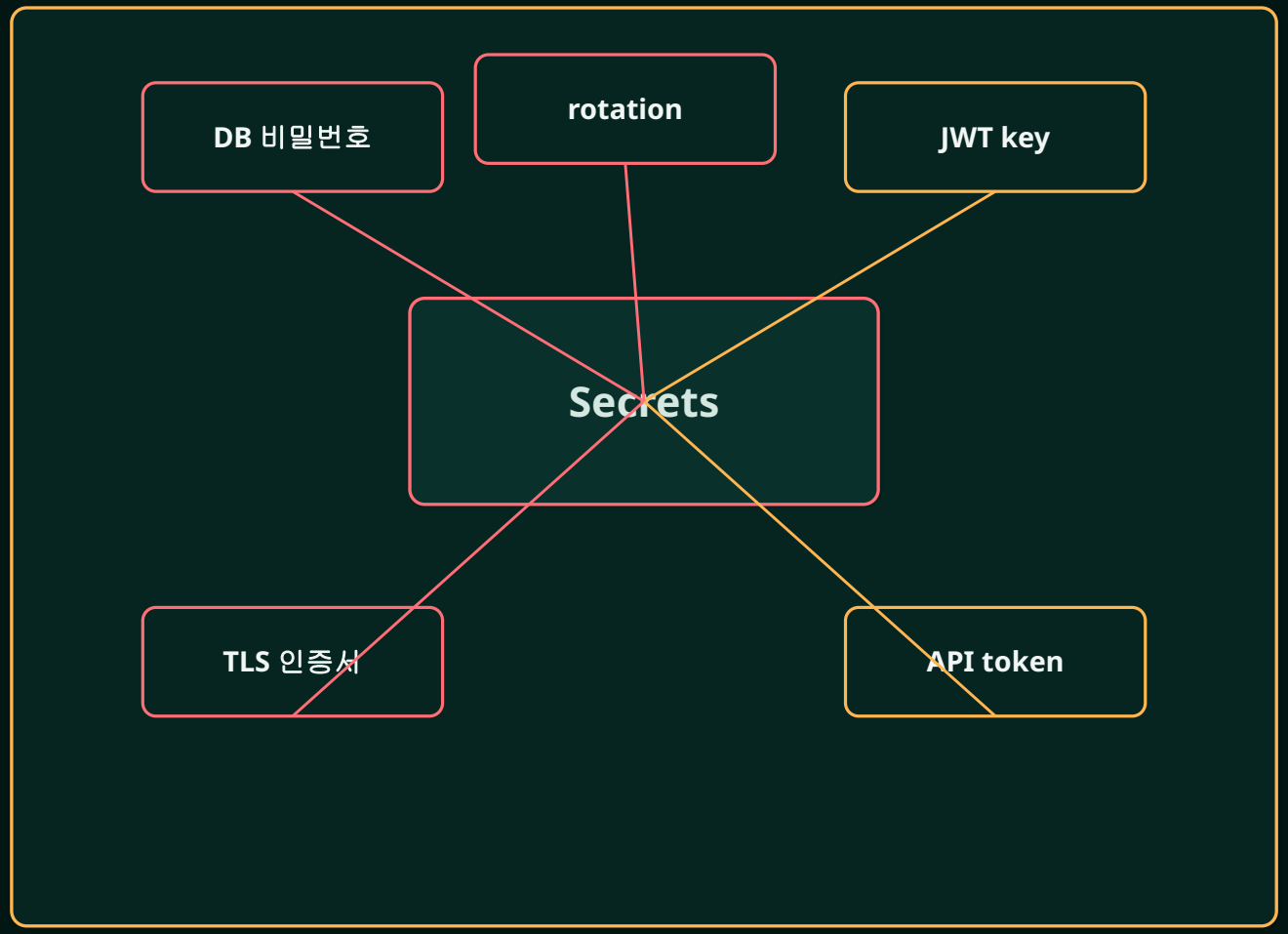
코드 저장소, 이미지, 로그, 채팅방에 비밀이 남지 않게 한다

02

중앙 비밀 저장소, 최소 권한 접근, 회전, 만료, 감사가 필요하다

03

JWT 시대에는 서명 키, 세션 시대에는 저장소 자격증명이 핵심 비밀이 된다



로그·감사·탐지는 마지막 방어선이다

01

성공/실패 로그인, 권한 거부, 비정상 빈도, 관리자 행위를 구조화 로그로 남긴다

02

하지만 원문 토큰, 세션 ID, 주민번호 등 민감정보는 남기지 않는다

03

탐지는 예방을 대체하지 않지만, 사고 축소와 포렌식에는 필수다

Go Layered Architecture

Go 구조와 선택

01

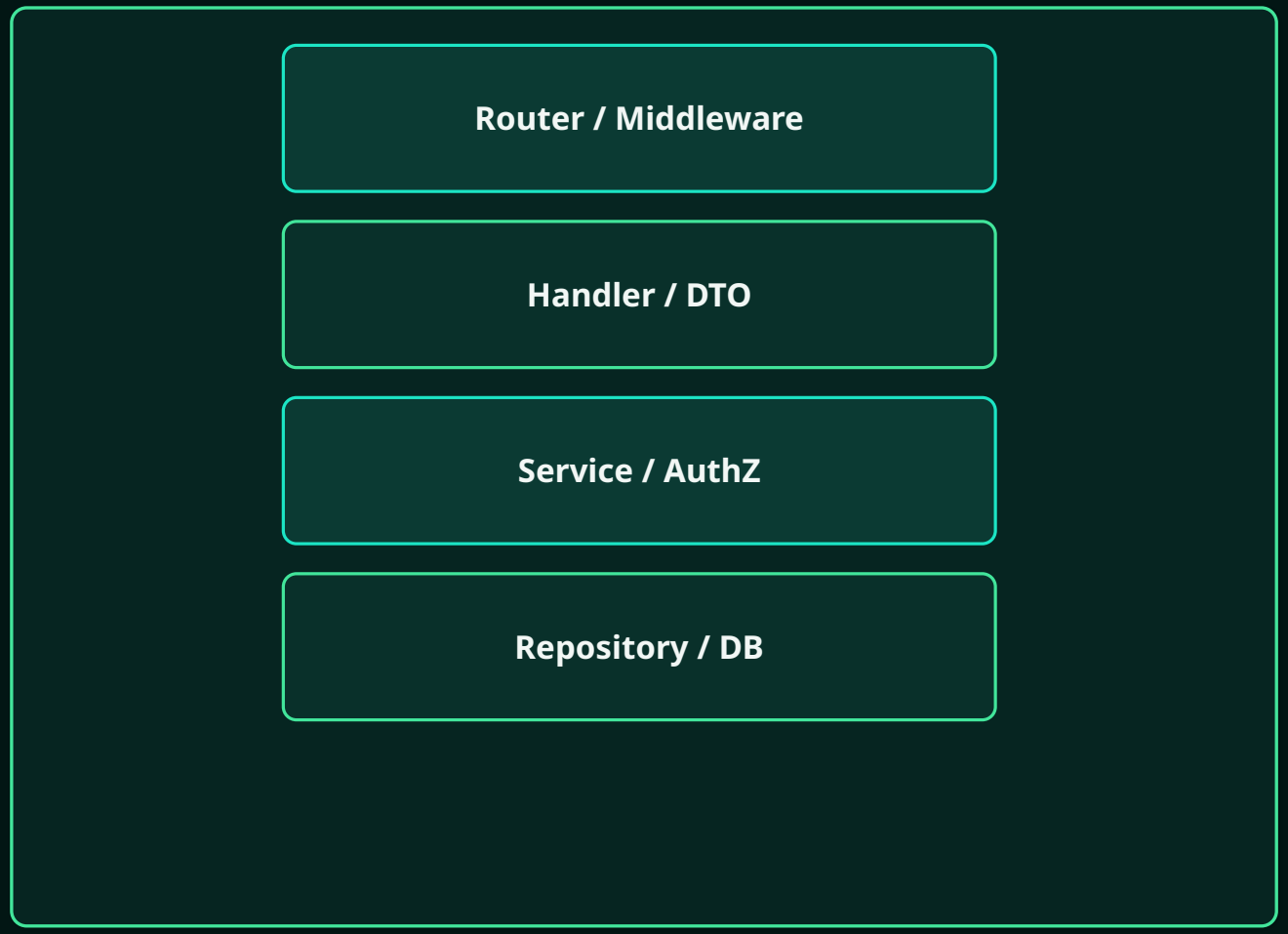
Handler: I/O, 파싱, 응답 형식

02

Service: 비즈니스 규칙, 인가, 상태 전이

03

Repository: 영속성, 쿼리, 트랜잭션



미들웨어, 서비스, 저장소의 책임 분리

01

인증(AuthN)은 미들웨어에서 컨텍스트를 주입

02

인가(AuthZ)는 서비스에서 행위 기준으로 판단

03

저장소는 이미 좁혀진 조건만 실행하고, 권한 결정을 대신하지 않는다

Middleware

01

인증 / 공통 정책

02

요청 컨텍스트

Service / Repository

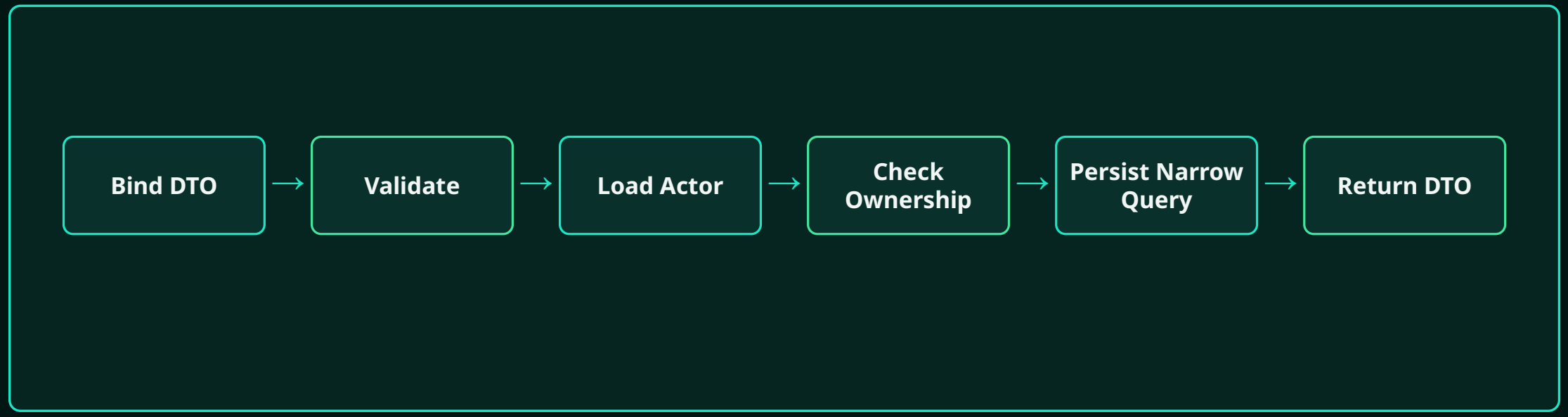
01

비즈니스 규칙 / 인가

02

저장소 쿼리

예시 요청 흐름: 안전한 수정 API



테스트 전략도 보안 구조를 따라간다

Go 구조와 선택

01

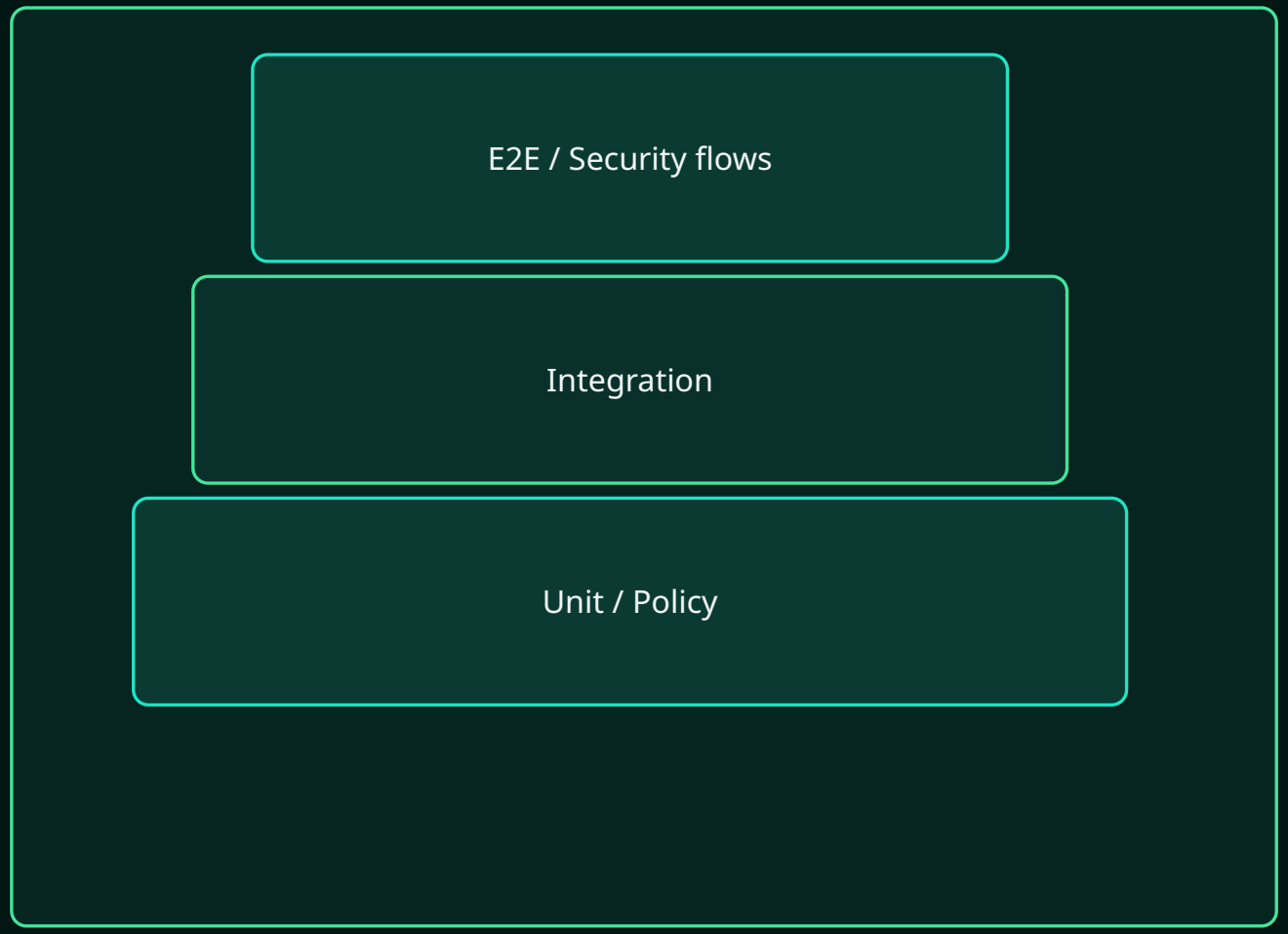
단위 테스트: validator, policy, permission matrix

02

통합 테스트: 미들웨어 + DB + 트랜잭션 + 인가 흐름

03

퍼징/경계값 테스트: 파서, URL, JSON, 문자열 길이, 정규식 입력



선택 매트릭스: 세션 vs 공유 세션 vs JWT vs 외부 IdP+BFF

기술 우열표가 아니라 운영 비용 구조 비교표

	세션	공유 세션	JWT	외부 IdP + BFF
주 사용처	same-domain 웹	여러 노드 웹	SPA·mobile·API	웹+모바일+SSO
상태 위치	앱 메모리/DB	중앙 저장소	토큰 내부	IdP + BFF 조합
즉시 회수	강함	강함	약함	중간~강함
운영 복잡도	낮음	중간	중간	높음
주요 대가	scale-out, CSRF	store 가용성	revocation, key	외부 의존, 복잡성

시나리오 1: 사내 관리자 시스템

Go 구조와 선택

01

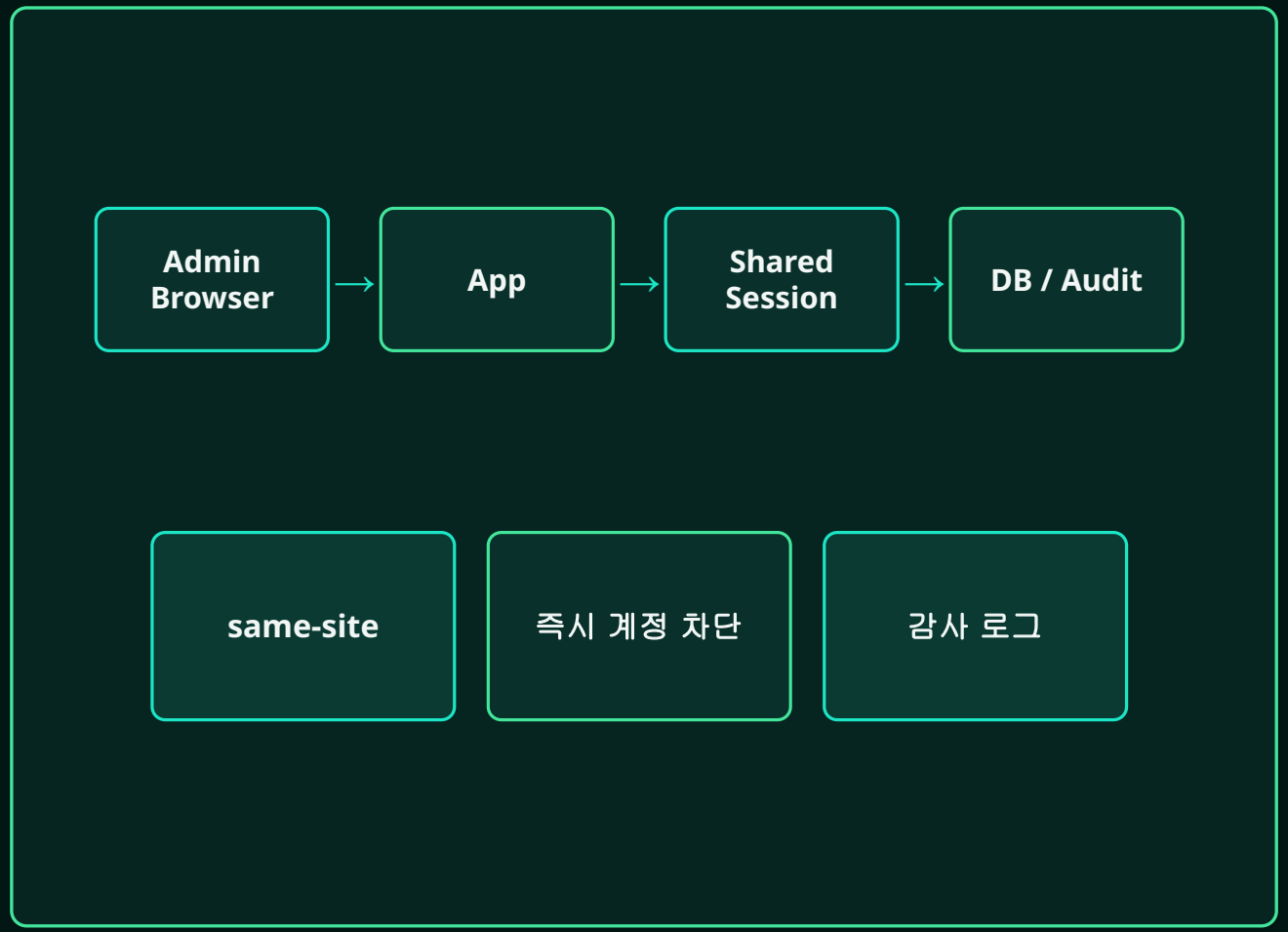
브라우저 same-site, 소수 사용자, 즉시 계정 차단 중요

02

권장: 공유 세션 + HttpOnly cookie + 강한 인가 + 감사 로그

03

외부 IdP는 조직 규모와 계정 관리 요구가 있을 때만 추가



시나리오 2: 소비자용 웹 + 모바일 동시 운영

01

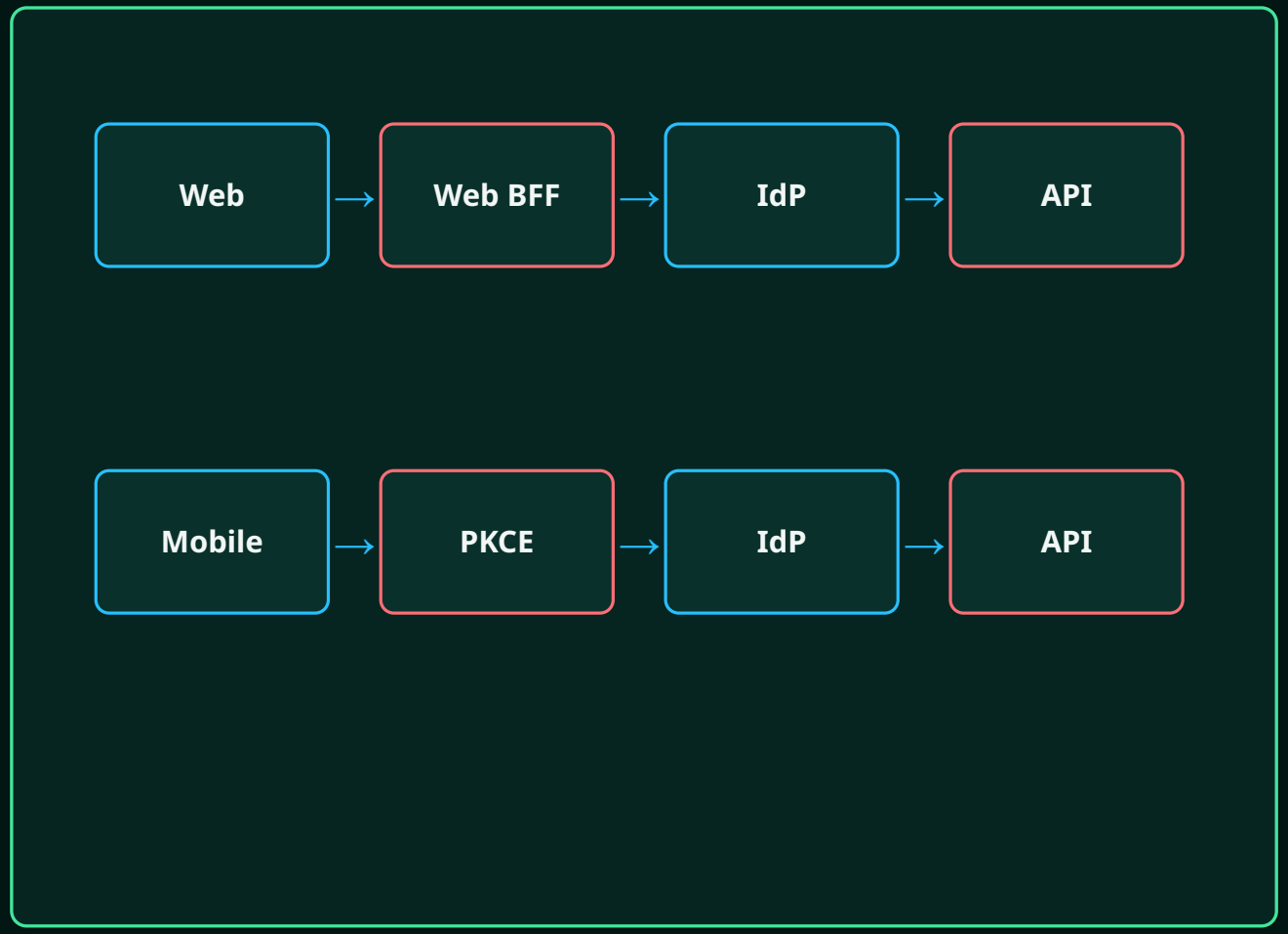
웹과 모바일의 저장 전략과 UX가 다르므로 동일 패턴 강요가 어렵다

02

권장: 외부 IdP/OIDC + 웹은 BFF cookie, 모바일은 PKCE 기반 토큰

03

공통 정책은 IdP에, 클라이언트별 표현과 경계는 BFF에 둔다



시나리오 3: 파트너 API와 서드파티 연동

Go 구조와 선택

01

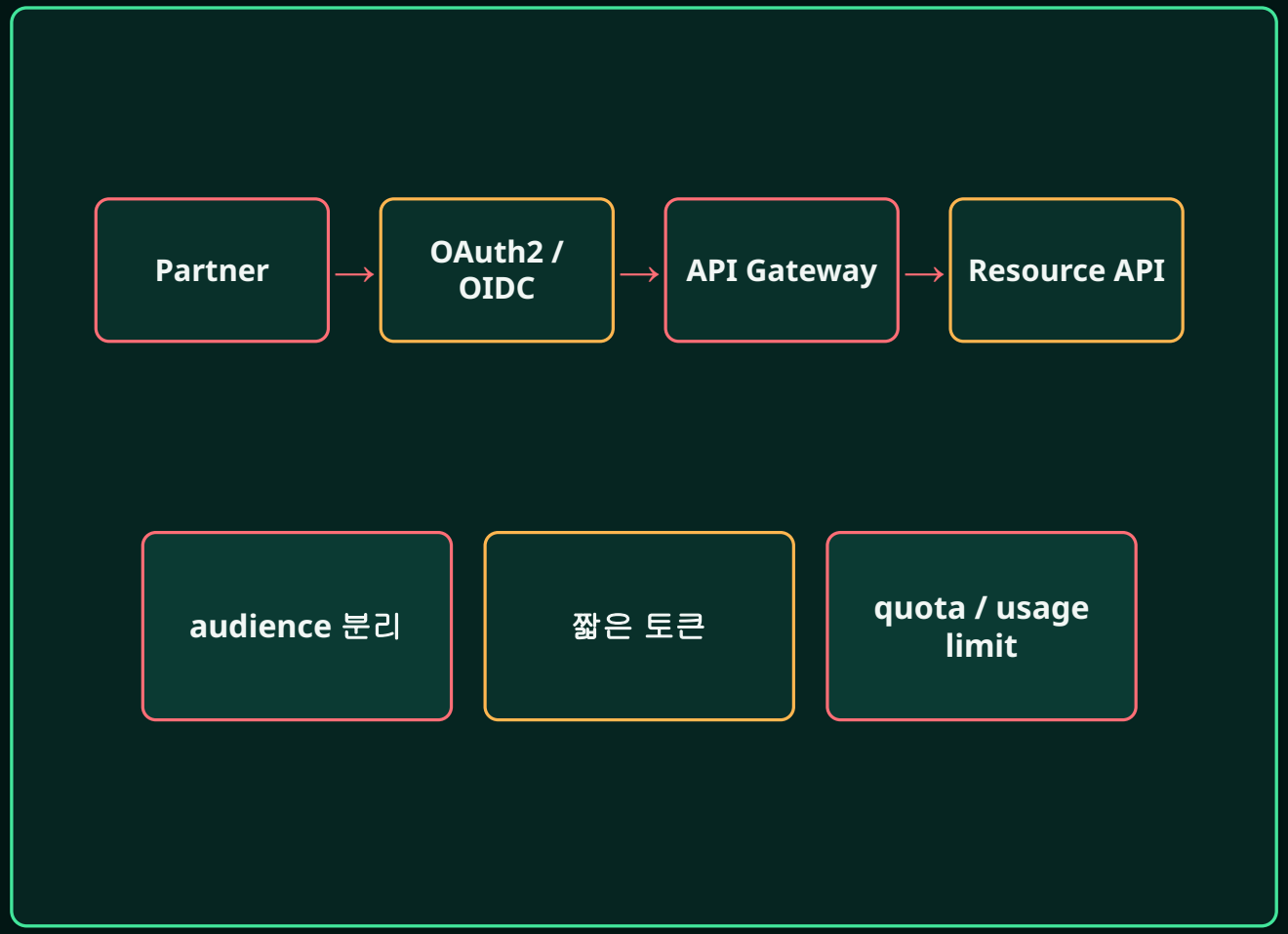
브라우저 세션보다 토큰 기반 위임과 scope 관리가 더 중요하다

02

권장: OAuth2/OIDC, audience 분리, 짧은 토큰 + 회전 + 사용량 제한

03

민감 API는 token introspection 또는 더 강한 송신자 구속까지 검토한다



최종 체크리스트: 선택을 명시적으로 하라

01

입력은 무엇을 허용할 것인가?

02

상태는 어디에 둘 것인가? 즉시 회수는 필요한가?

03

출력은 누구에게 어디까지 보여 줄 것인가?

04

자원은 누가 얼마나 쓰게 할 것인가?

입력 허용 범위

상태 위치 / revoke

출력 최소화

자원 상한

키 / 비밀 관리

로그 / 감사

기술 발전은 신뢰 경계의 이동이다

Go 구조와 선택

01

세션에서 JWT로, 내부 인증에서 외부 IdP로 갈수록 문제가 사라진 것이 아니다

02

입력·상태·출력·자원의 부담이 다른 위치로 이동했을 뿐이다

03

좋은 설계는 최신 기술을 쓰는 것이 아니라, 그 이동 비용을 이해하고 선택하는 것이다