KSUG SPRING CAMP 2024

구해줘 홈즈!

은행에서 3천만 트래픽의 홈 서비스 새로 만들기

카카오뱅크 홈서비스개발팀
이영규 tigger.lee

발표자 소개

- 카카오뱅크 홈 서비스 개발팀
- 주로 홈과 이체 관련 서비스를 개발
- 경력 만 5년 6개월

Overview

- 은행에서의 홈 서비스 분리 여정을 소개
- 크게 2가지 주제
  - 분리하며 기술 부채를 해결한 이야기
  - 서비스를 안정적으로 이관한 이야기

- 시간 관계 상 배경 지식 설명은 최소화
  - 헥사고날 아키텍처, DDD, 코루틴
- 편하게 들어주세요

이관 배경

기술부채 해결

안정적 이관 전략

결과

이관 배경

기술부채 해결

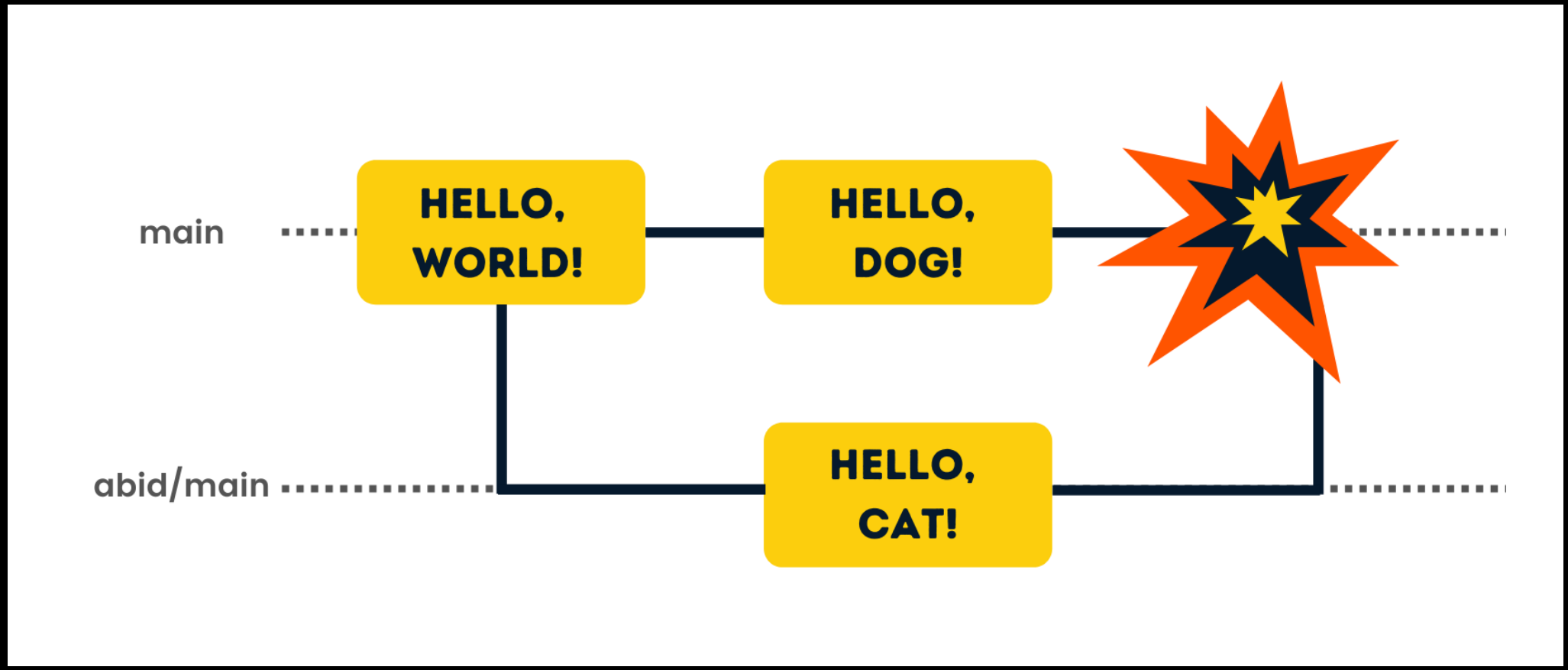
안정적 이관 전략

결과

Legacy Monster

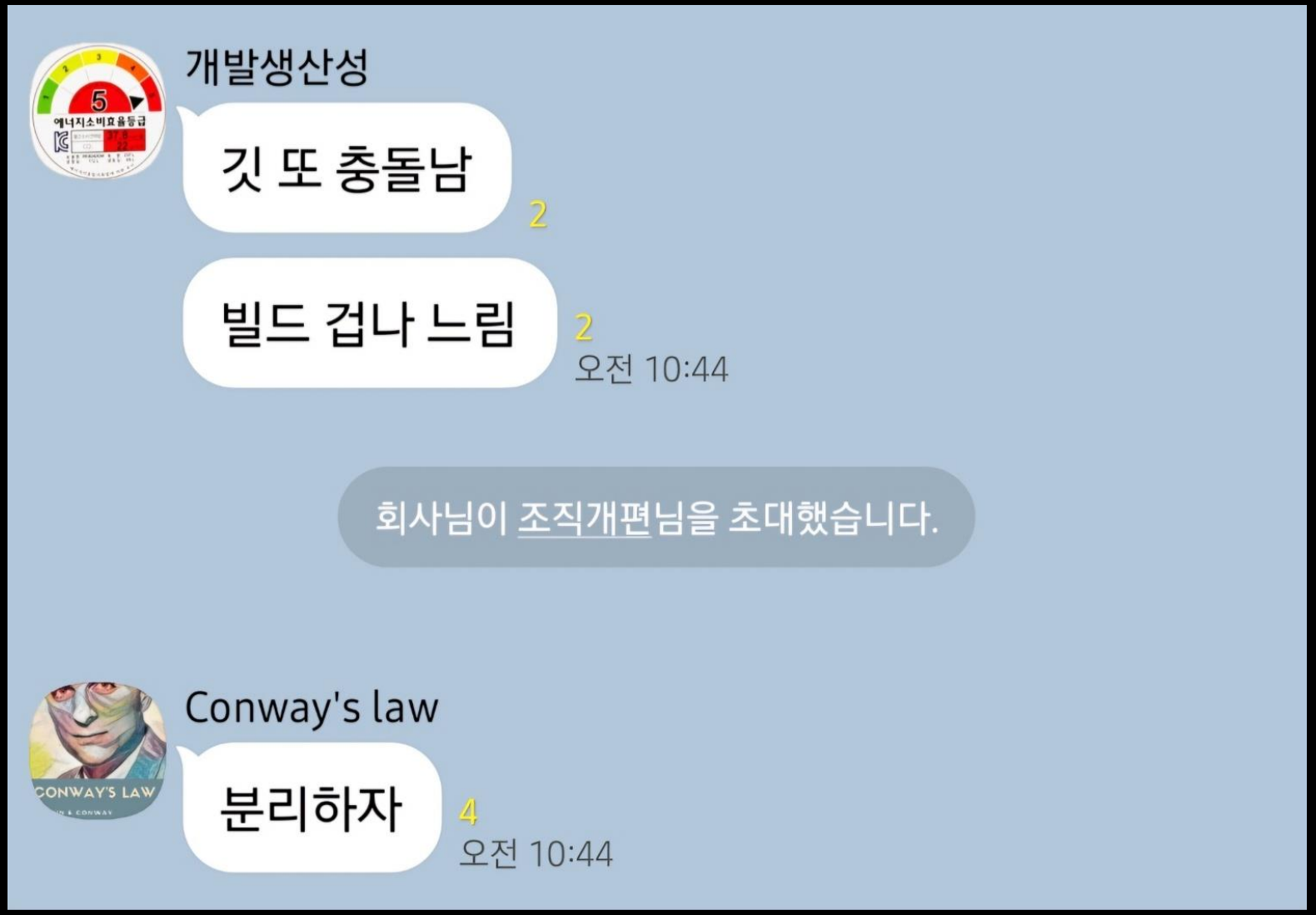
Ref) Apple TV+, Monarch : Legacy of Monsters

떨어지는 개발 생산성

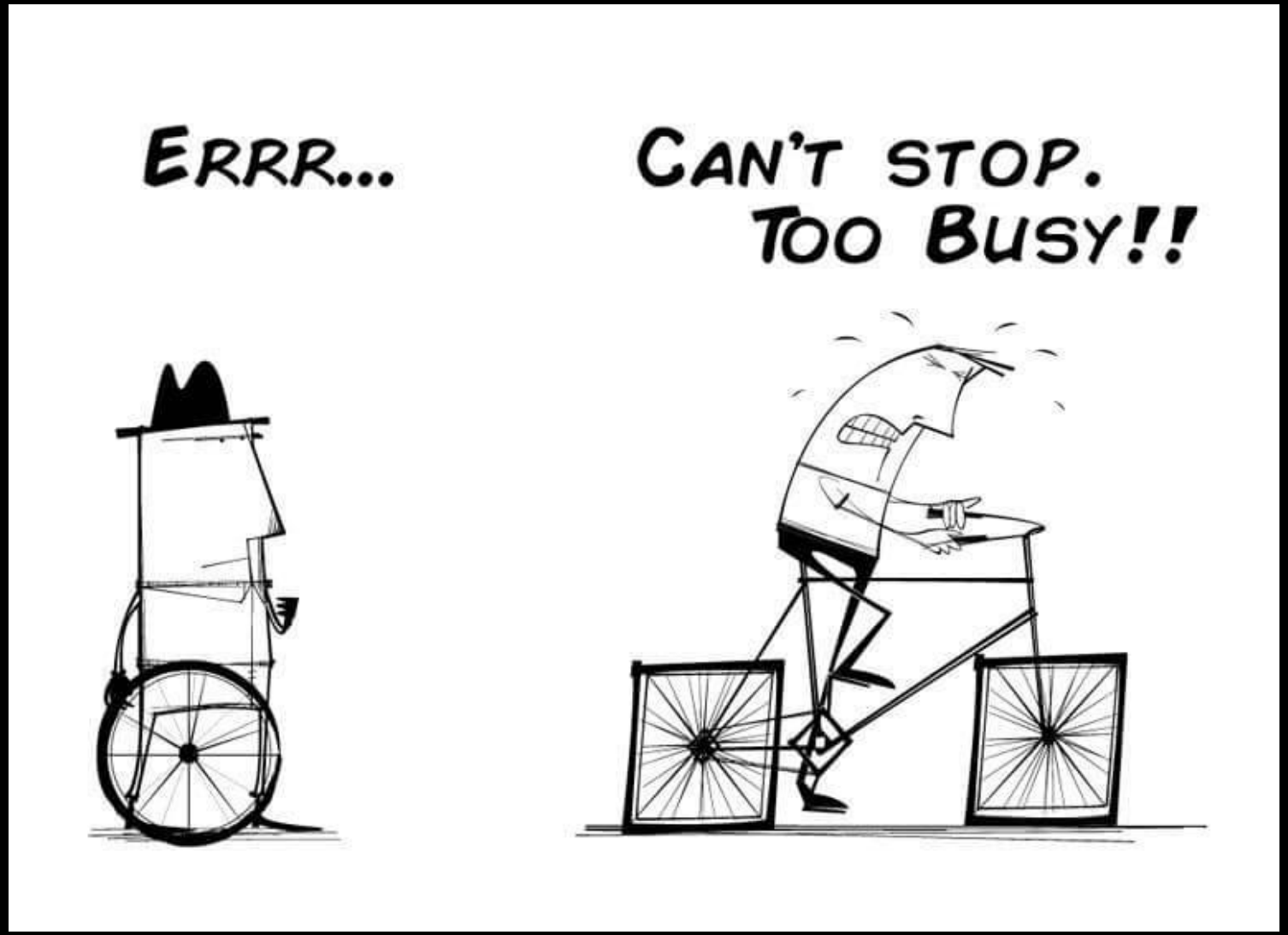


Ref) Data Camp, How to Resolve Merge Conflicts in Git Tutorial

It’s Time to Migration!



다른 기술 부채도 갚아볼까?



Ref) Gata Software, Technical Debt : Bad for Your Business

해결하고자 했던 문제들

1) 구조적 문제
   - 계층 간 의존성이 꼬여 있음
   - 외부 의존성과 도메인 정책이 혼재되어 섞여 있음

2) 성능적 문제
   - 호출해야 하는 외부 서비스 증가에 따른 성능상 우려

하지만 확보해야 하는 안정성

Ref) Unews Archive, Study Shows Money Cues Can Trigger Unethical Behavior, University of Utah, June 20, 2013

서비스를 이관하면서, 문제도 해결하는데, 안정적으로

Ref) 무리한 요구를 하는 클라이언트, 카카오톡 이모티콘

이관 배경

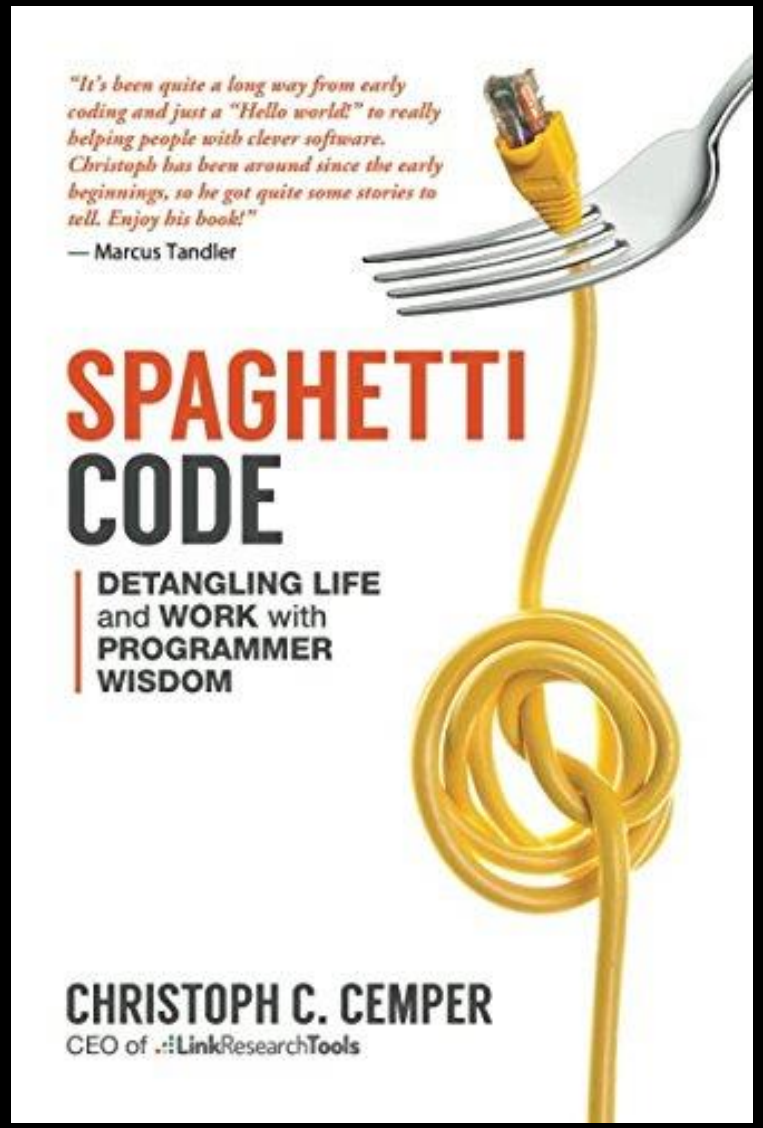
기술부채 해결 – (1) 구조적 문제

안정적 이관 전략

결과

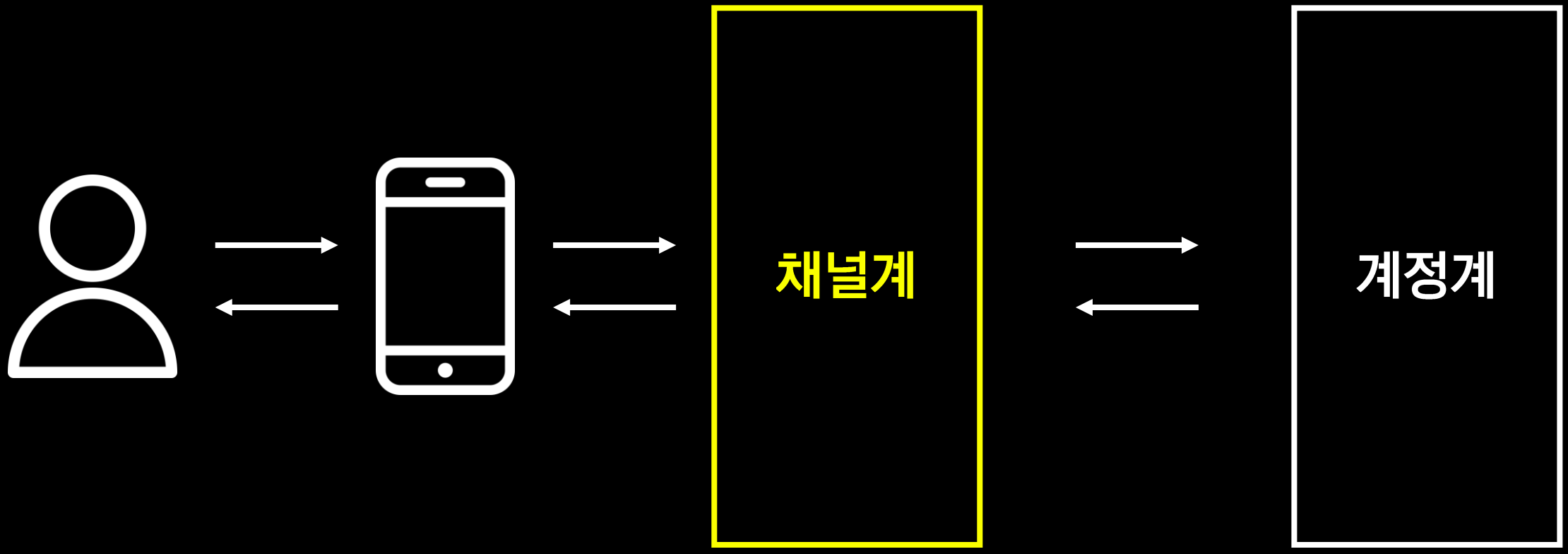
구조적 문제 : 섞여 있는 외부 의존성과 도메인 정책

1) 구조적 문제
   - 계층 간 의존성이 꼬여 있음
   - 외부 의존성과 도메인 정책이 혼재되어 섞여 있음

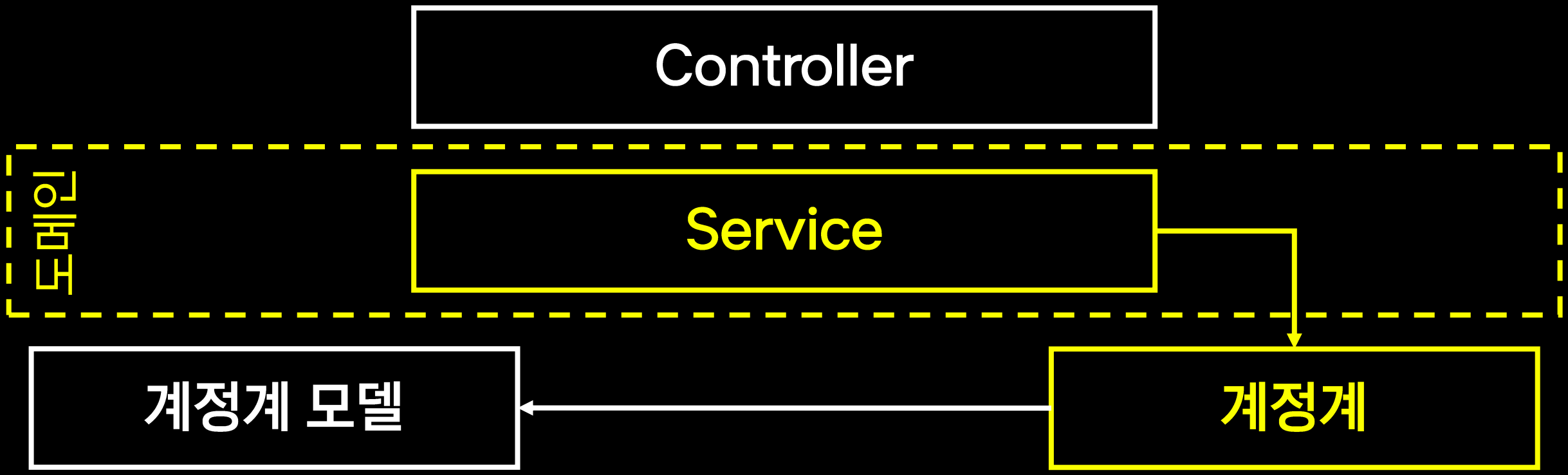


Ref) Christoph C. Cemper, Spaghetti Code: Detangling Life and Work with Programmer Wisdom

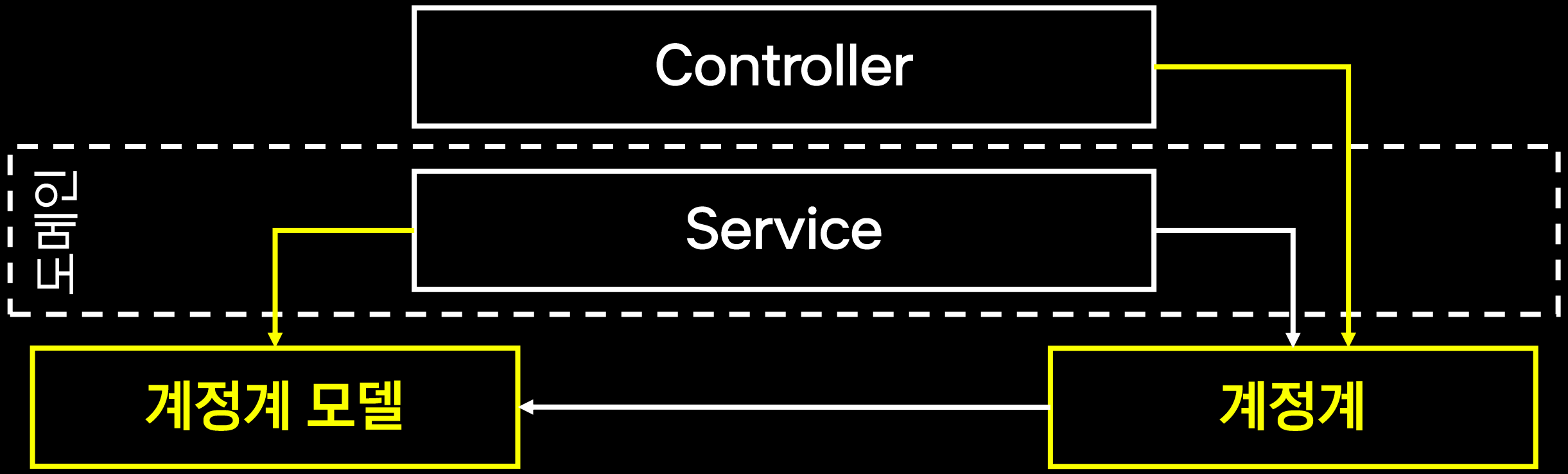
계정계? 채널계?



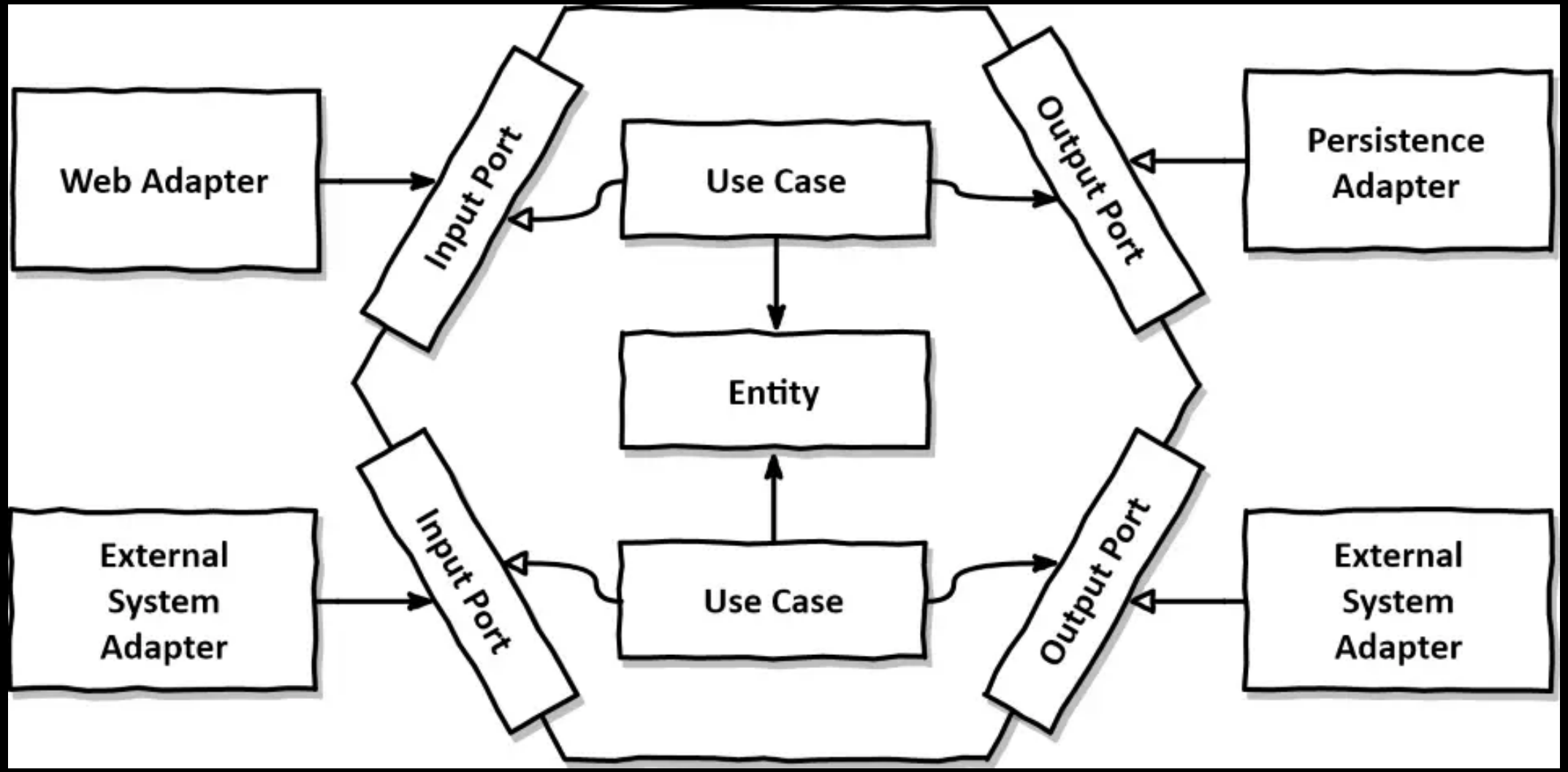
계정계를 의존하는 도메인 Service



계정계 Driven Development



Hexagonal Architecture ?



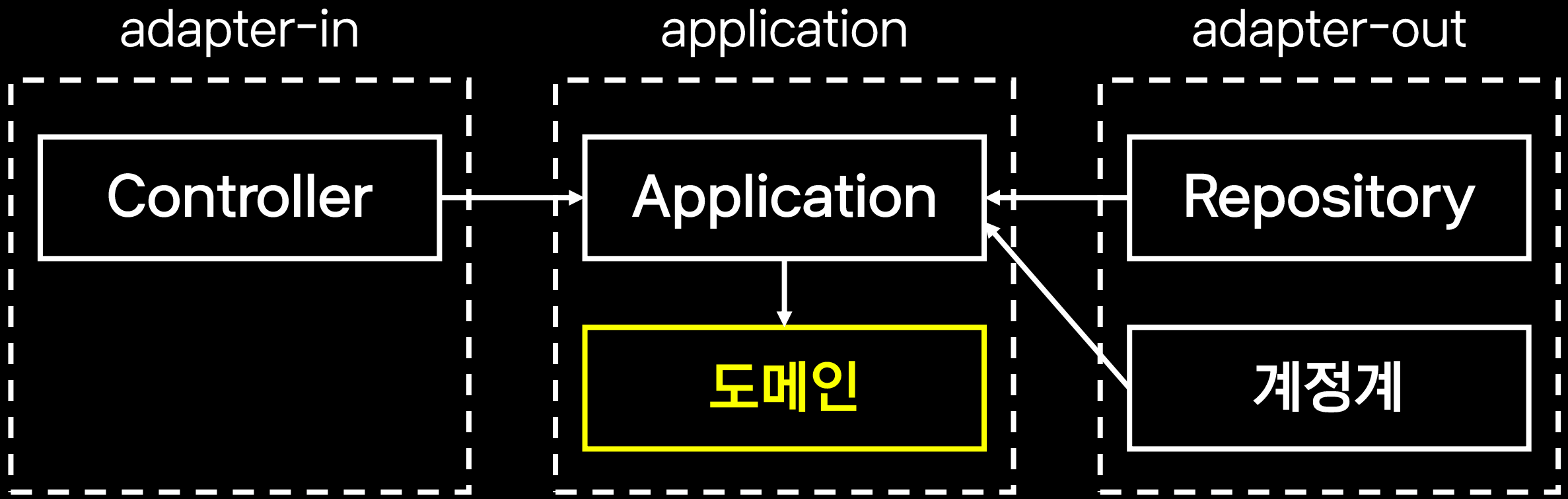
Ref) Get your hands dirty on Clean Architecture

Hexagonal Architecture 의 특징

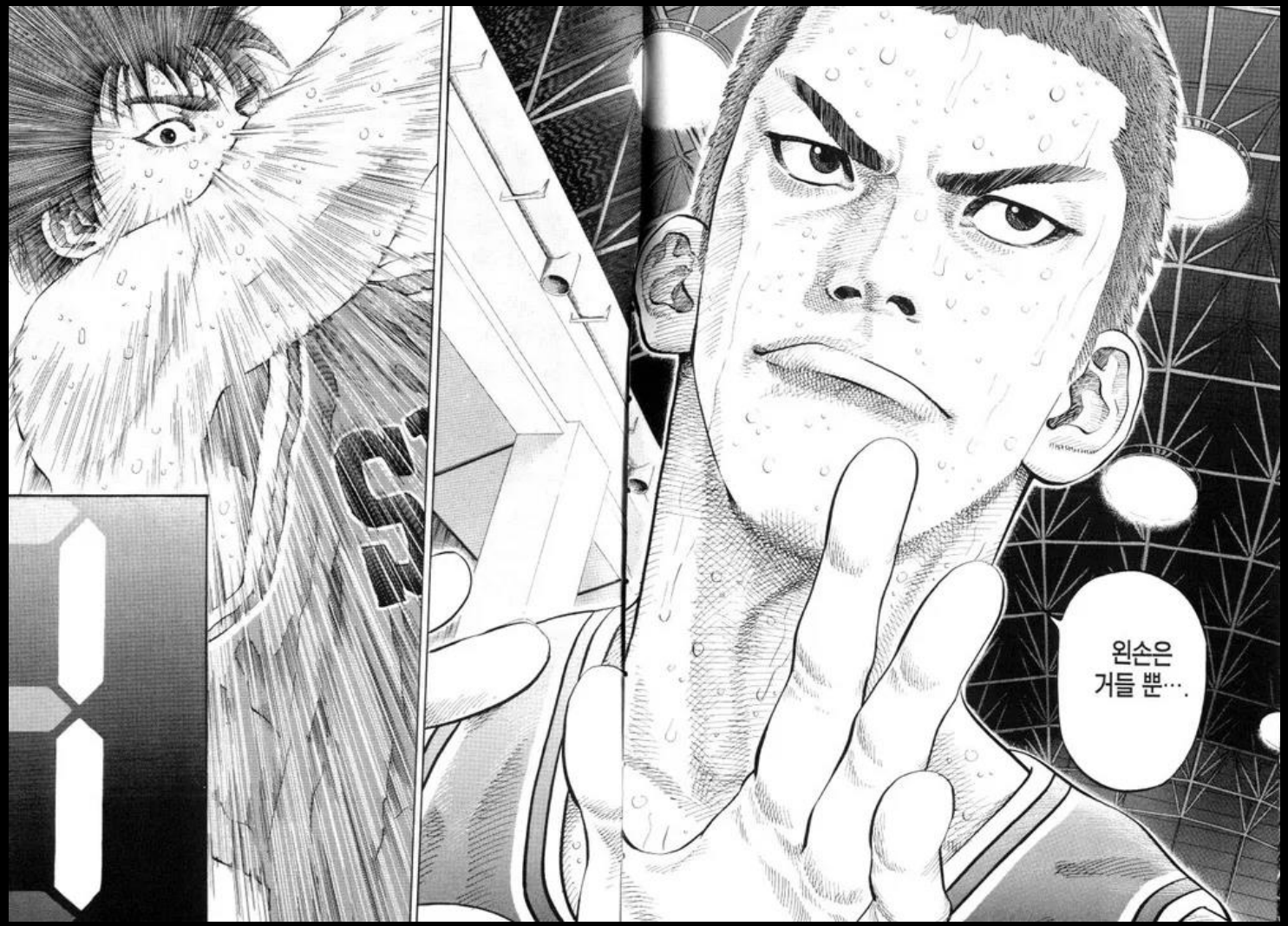
- 지름길을 택하지 않게 된다 (택할 수 없게 된다)
- 의존성의 방향과 도메인 계층의 보호를 강제한다
- 유즈 케이스가 좀 더 가시적으로 드러난다
- 동시 작업이 쉬워진다

- 보일러 플레이트 코드가 많아진다
- 초기 러닝 커브가 존재한다

Hexagonal Architecture 를 도입해보자



아키텍처는 거들 뿐

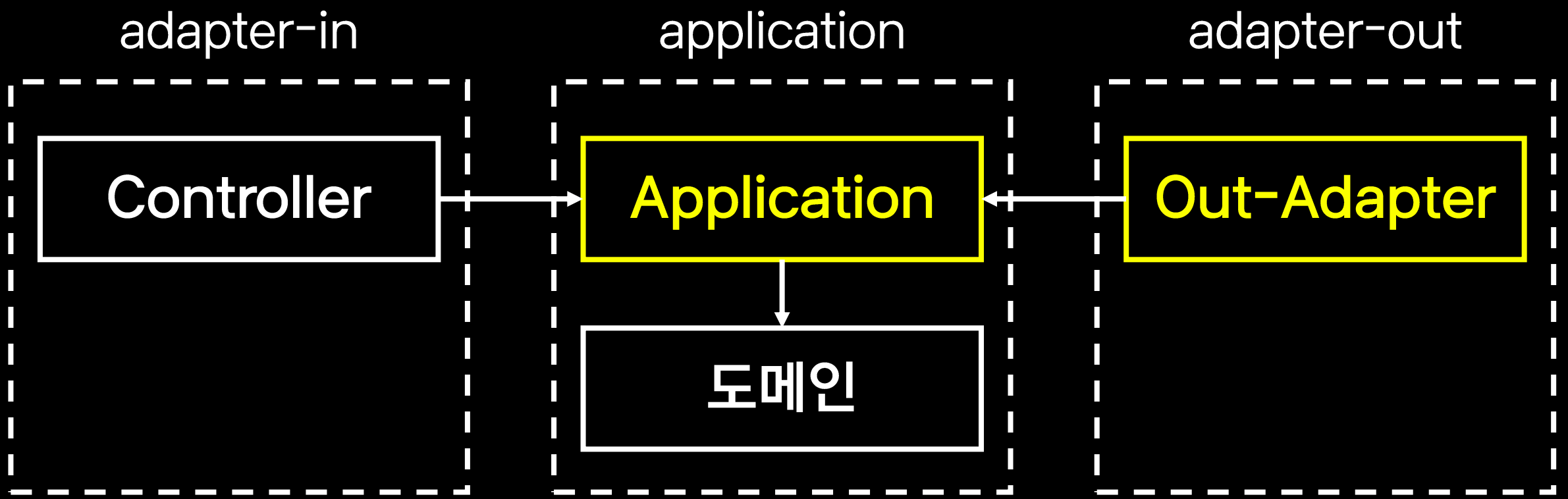


Ref) 이노우에 다케히코, 슬램덩크, 대원씨아이

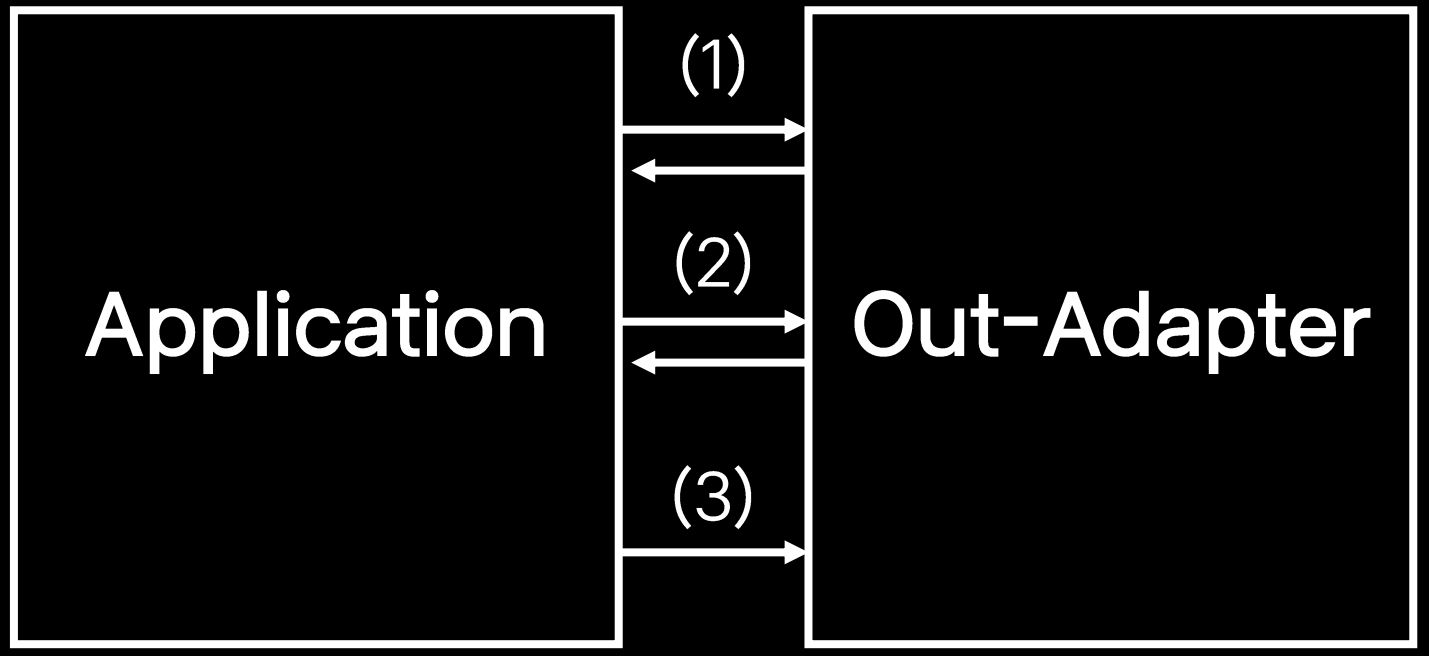
어디로 가야하오

Ref) 대중문화부·게임분석팀 기자, “[오늘의 유머] 리그 오브 레전드(롤), 리신 실사판은 어떤 모습?”, 엑스포츠뉴스, 2013.10.15

Application vs. Out-Adapter



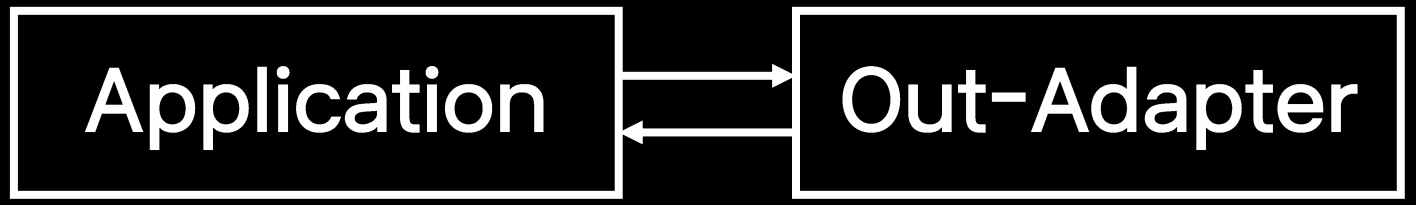
고민 사례 1: 외부 응답 캐시



(1) 캐시를 조회하고

(2) 캐시가 없으면,
외부 서비스 조회

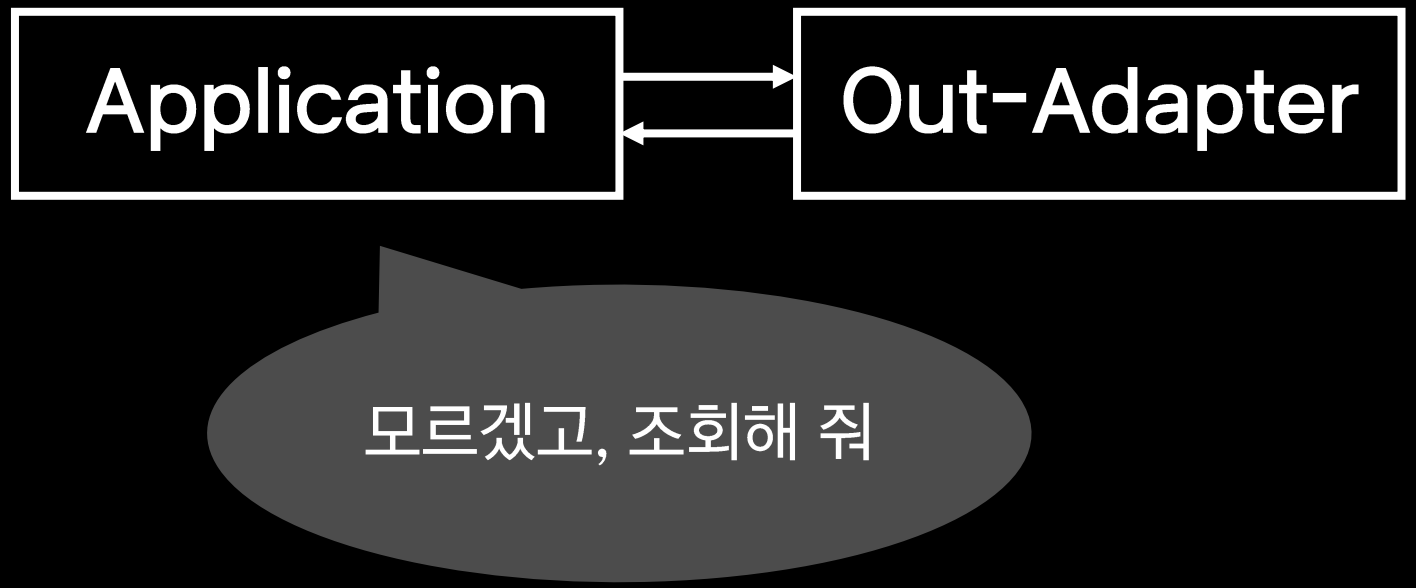
(3) 캐시 저장



알아서 캐시 조회/갱신/응답

비즈니스 정책인가? 선택적 옵션인가?

- 캐시가 중요한 비즈니스 정책인가?
- 항상 캐시해야 하는 데이터인가?



반례: 오픈뱅킹 잔액 조회 사례

- 오픈뱅킹 잔액은 조회할 때마다 비용이 발생
- 은행 별 트래픽 제한도 존재

화면1

잔액이 중요한 화면

잔액이 정확해야 함

캐시 X

화면2

잔액이 중요하지 않은 화면

빠른 응답이 더 중요함

캐시 O

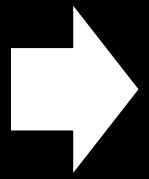
* 오픈뱅킹 : 고객이 다른 금융기관에 등록한 데이터를 공유하는 서비스

* 오픈뱅킹 잔액 : 다른 은행 계좌의 잔액

고민 사례 2: 외부 서비스 응답의 어그리게이션

계좌

- 잔액
- 거래내역
- 제휴 정보
- 별칭(이름)



```
data class Account(
  val balance: Balance,
  val activities: List<Activitiy>,
  val partnershipInfo: PartnershipInfo,
  val alias: String,
)
```

```
interface LoadAccountPort {
  fun loadAccount(id: AccountId): Account
}
```

서로 다른 곳에서 조회해야 한다면?

계좌

- 잔액 => 계정계
- 거래내역 => 계정계
- 제휴 정보 => 외부 서비스
- 별칭(이름) => RDB

외부 호출 및 조합은 누구의 역할일까?

LoadAccountPort

LoadBalancePort

LoadActivitiesPort

LoadPartnershipPort

LoadAliasPort

도메인의 경계를 어떻게 나눌 것인가

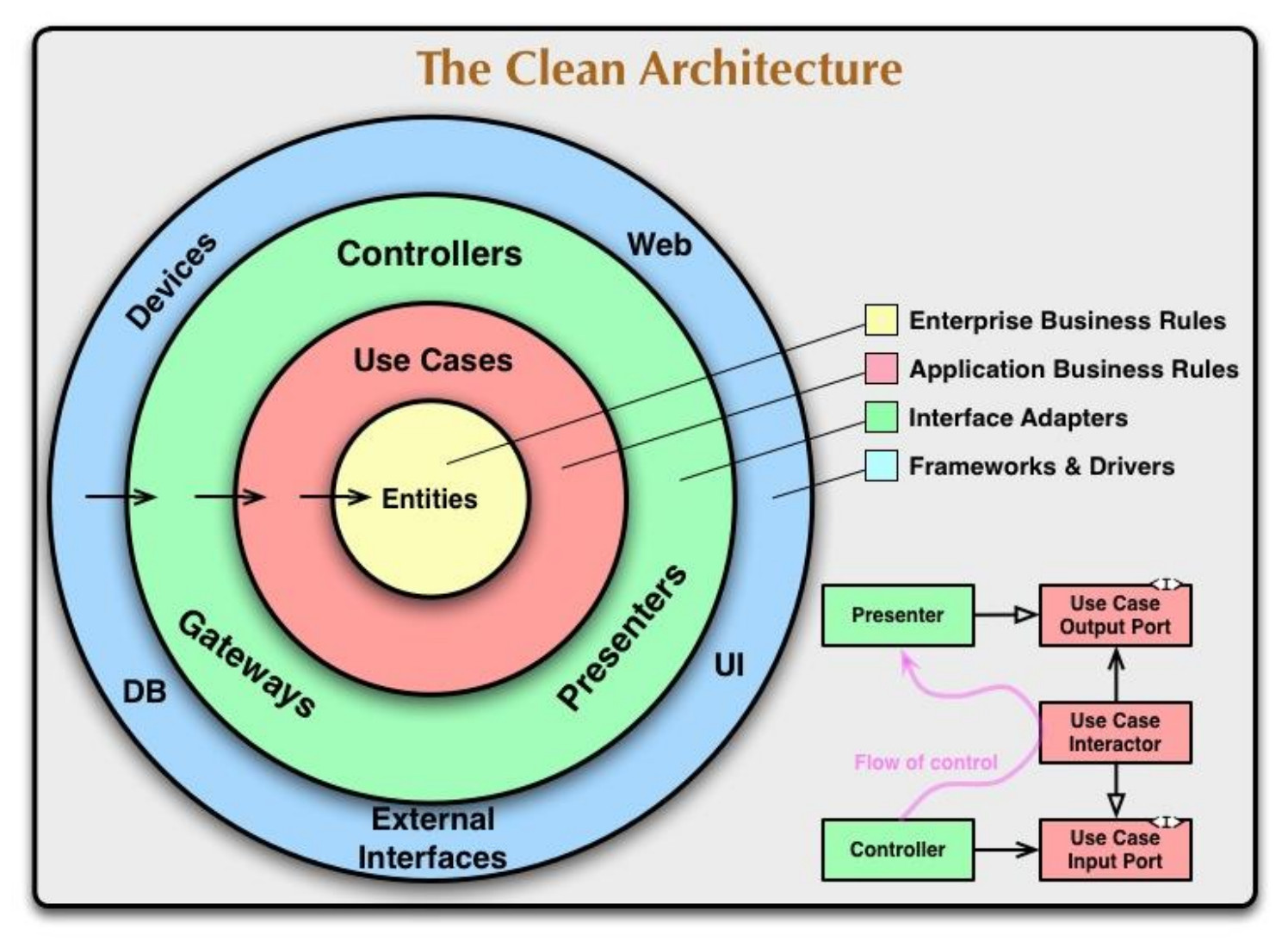
```
data class Account(
  val balance: Balance,
  val activities: List<Activitiy>,
  val partnershipInfo: PartnershipInfo,
  val alias: String,
)
```

```
data class PartnershipInfo(
  // info
)
```

LoadAccountPort

LoadPartnershipPort

결국 중요한 건 도메인



Ref) Robert C. Martin, The Clean Architecture, The Clean Code Blog, 13 August, 2012

이관 배경

기술부채 해결 – (2) 성능적 문제

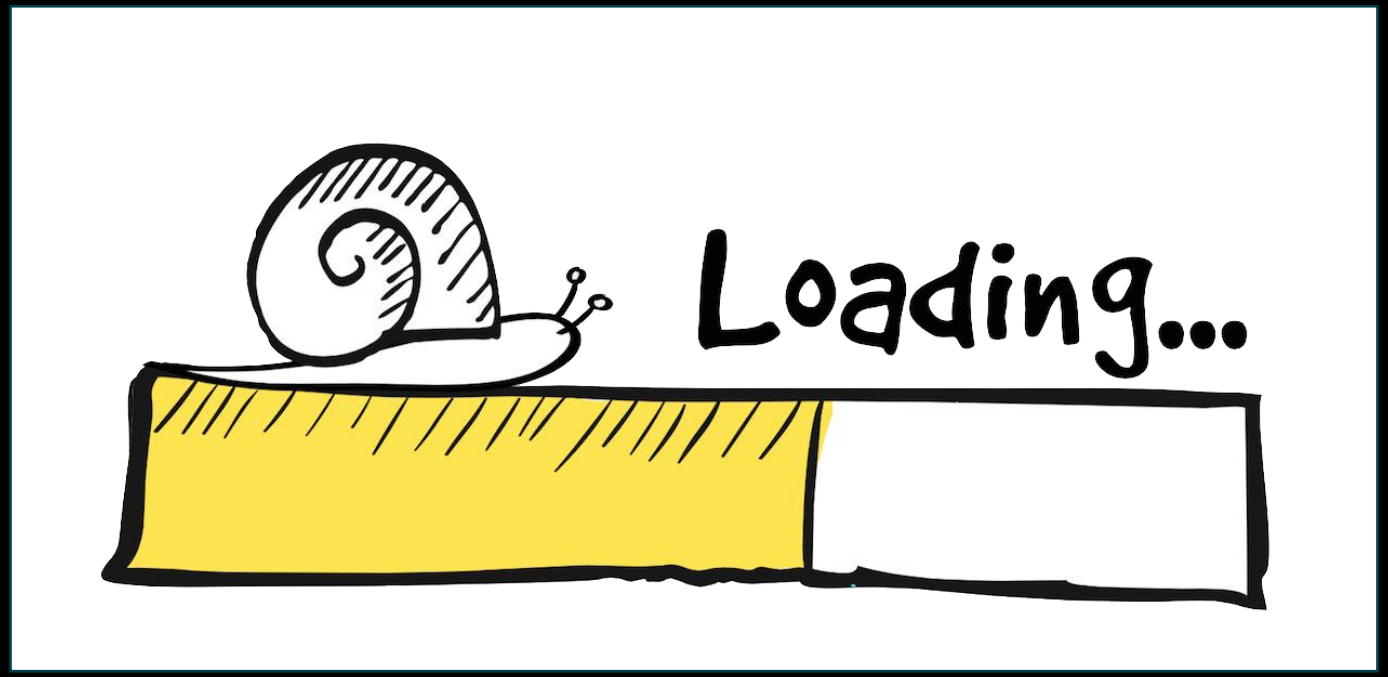
안정적 이관 전략

결과

성능적 문제 : 많은 외부 호출과 성능

2) 성능적 문제

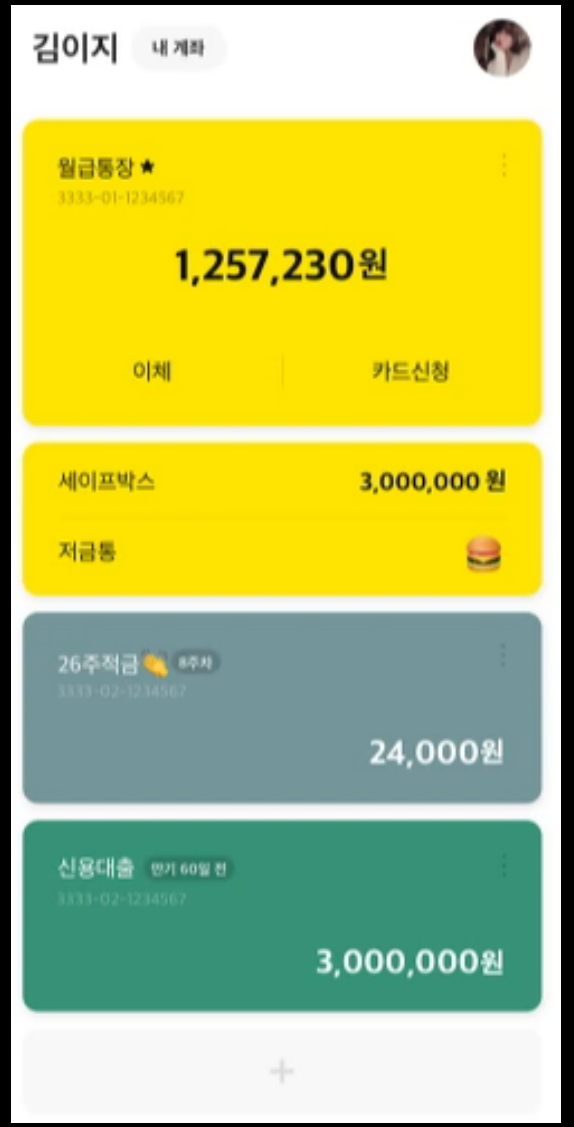
- 외부 서비스 호출 증가에 따른 성능상 우려



Ref) PNG.EGG, Snail loading chart, snails, loading png

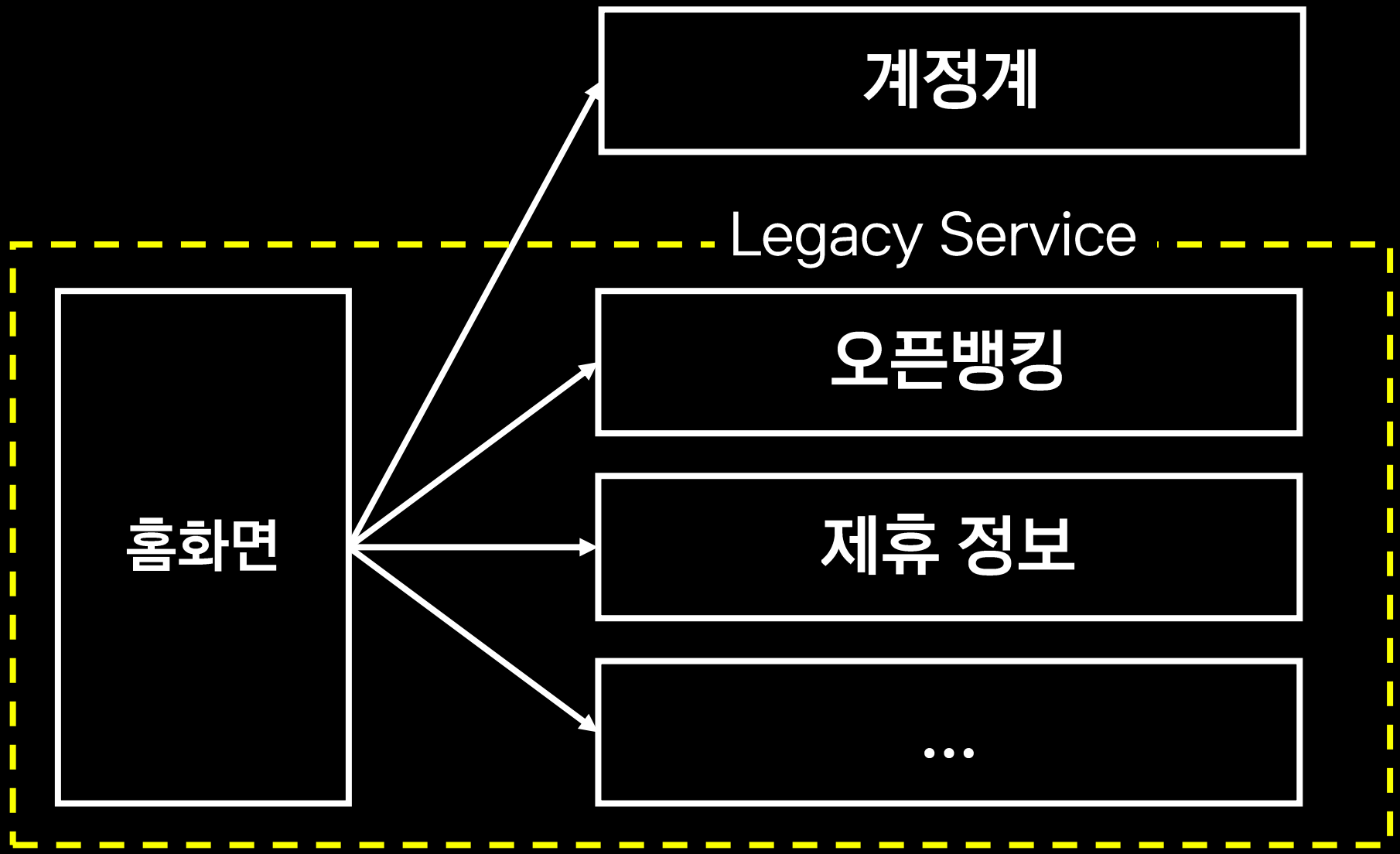
홈 화면?

- 여러 개의 계좌 목록
- 다양한 계좌 타입
- 각종 서비스 및 제휴 정보

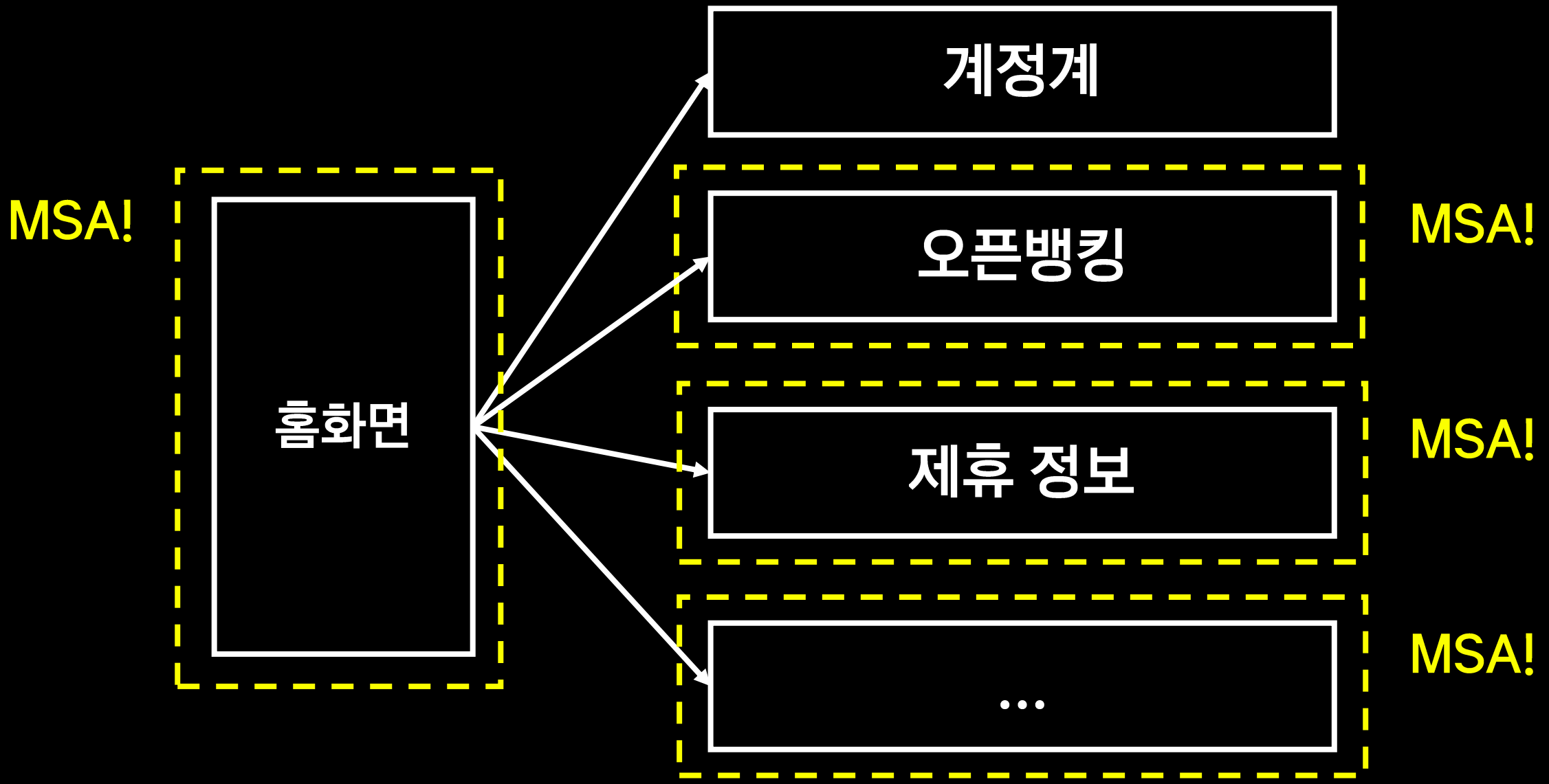


Ref) kakaocorp, 같지만 다른 은행, 카카오뱅크

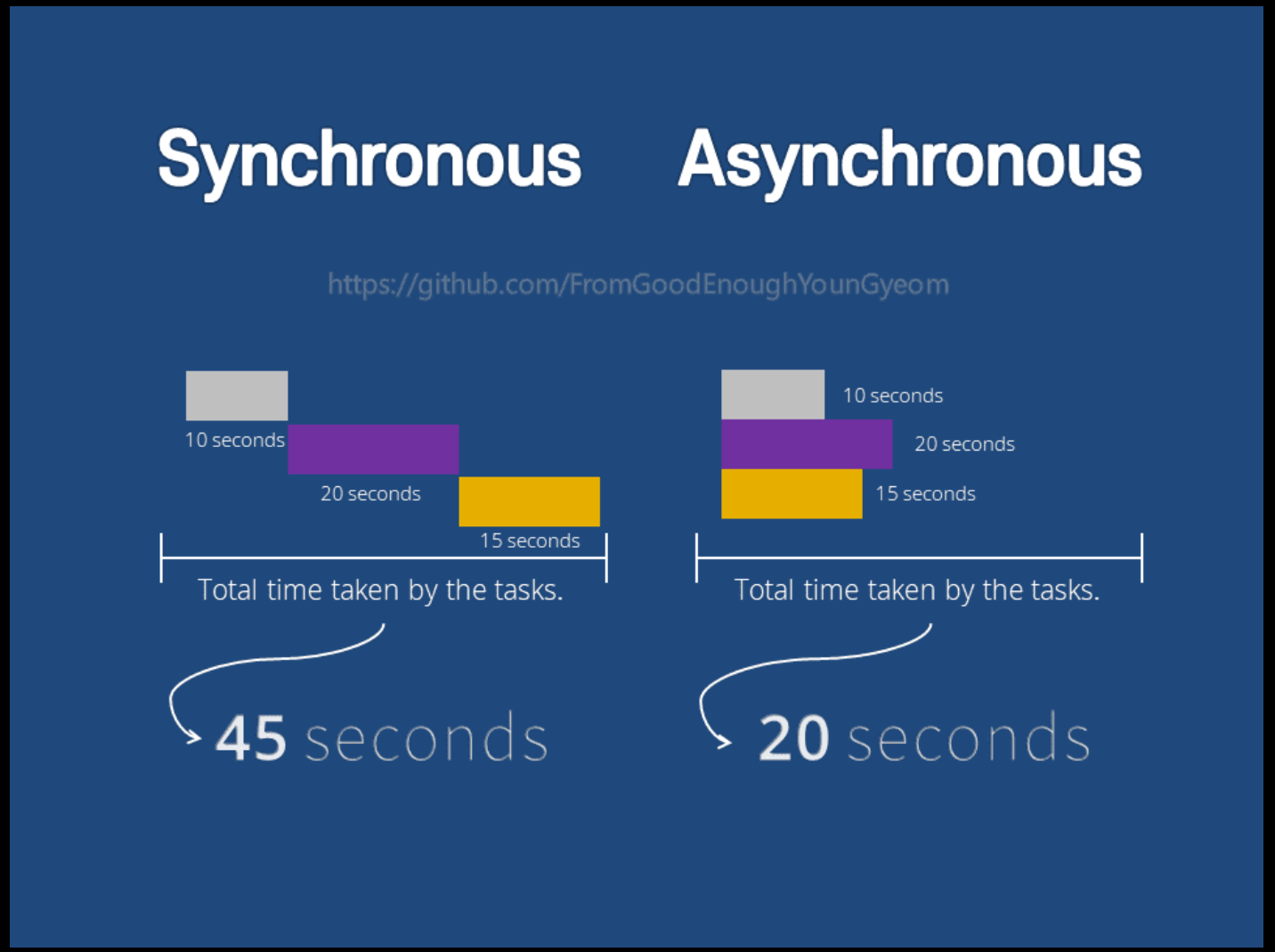
홈 화면의 특징



분리되는 서비스들, 길어지는 네트워크 구간



동시성을 통해 해결해보자



Ref) FromGoodEnoughYounGyeom, [Web Development] 동기(Synchronous),비동기(Asynchronous) 처리, January 01, 2021

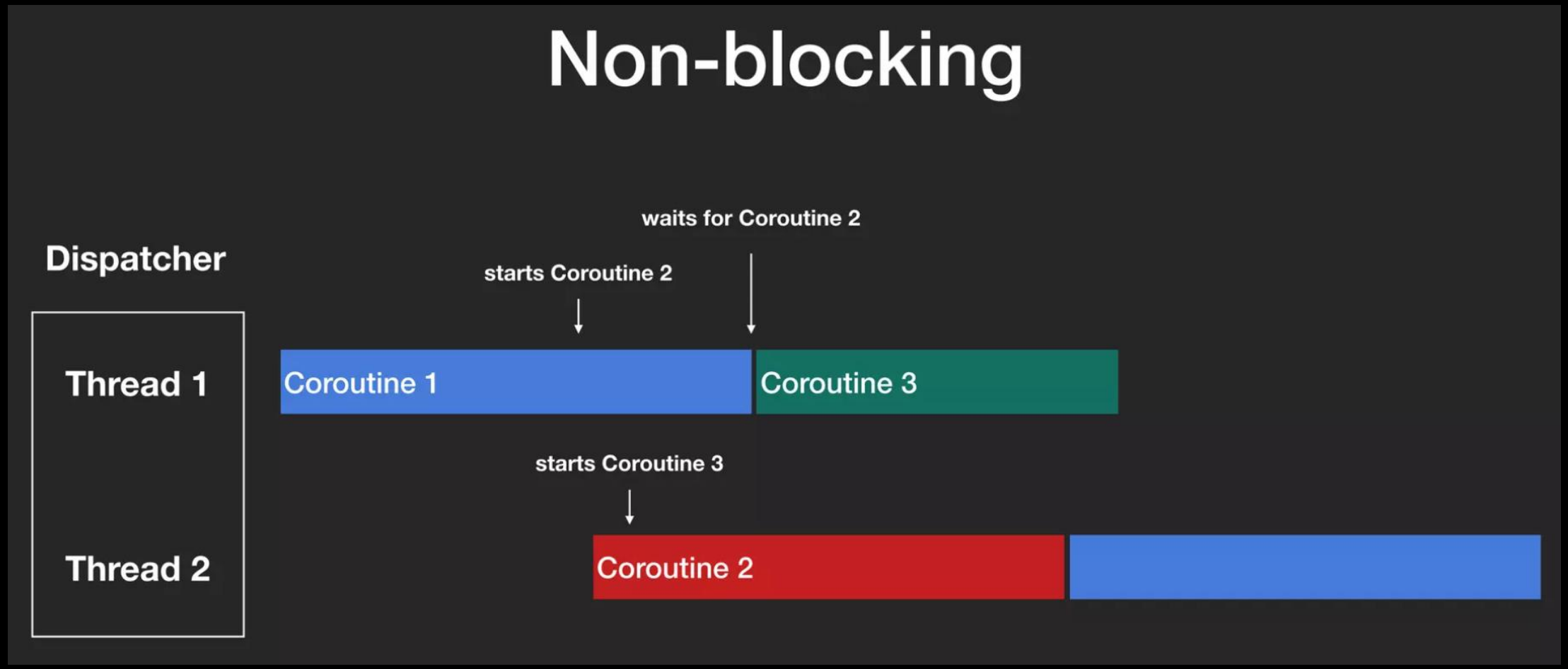
동시성을 도입하기 위한 3가지 선택지

- Spring @Async
- Spring Webflux
- Kotlin Coroutine

Webflux 의 안 좋은 추억

- 코드에 불필요한 동시성 관련 요소들이 많이 침투함
- Mono, Flux 등 복잡한 인터페이스를 따로 배워야 해서 러닝커브 존재
- 모니터링이 어렵고 레퍼런스가 부족

Kotlin Coroutine ?



Ref) Bart ł omiej Osma ł ek, Kotlin Coroutines - the new async, slideshare, Nov 22, 2018

간결한 코드 vs. Spring @Async

```
@Async
open fun doSomethingAsync(
): CompletableFuture<Res> {
  return CompletableFuture.completedFuture(Res())
}
```

```
suspend fun doSomethingSuspend(): Res {
  return Res()
}
```

```
fun useCase() {
  val res: CompletableFuture<Res>
    = doSomethingAsync()
  res.thenAccept { println(it) }
}
```

```
suspend fun useCase() {
  val res: Res = doSomethingAsync()
  println(res)
}
```

Kotlin Coroutine !

- 단순 멀티 스레드 방식보다 더 효율적으로 동작
- 다른 방식에 비해 훨씬 간결하게 사용할 수 있음
- 이미 Kotlin 언어를 사용하고 있는데, Kotlin 언어 수준에서 지원해 줌
- 안드로이드 개발자 분들이 이미 오래 전부터 도입해서 잘 사용하고 계심
- 새로운 기술 학습에 대한 열망

이 또한 쉽지 않은 여정

Ref) Shutterstock

Coroutine 을 도입해보자 - (1) suspend Controller

```
@GetMapping(“/hello”)
suspend fun hello(): Response<Something>
```

* suspend : 일시 중단 지점을 포함할 수 있다는 의미

- Controller 부터 계속 suspend 를 붙이면...
  - 별다른 빌더 없이 호출되는 모든 컴포넌트에서 중단함수를 사용 가능
  - i/o 작업이 많은 경우 코루틴에 의해 전체 시스템의 동작이 더 효율적
  - 호출되는 모든 컴포넌트 메서드에 suspend modifier 를 붙여야 함

전파되지 않는 thread local

DispatcherServlet.java

```
protected void doDispatch( ... ) {
  ...
  mv = ha.handle( ... )
  ...
```

InvocableHandlerMethod.java

```
protected Object doInvoke(Object... args) {
  ...
  if(KotlinDetector.isKotlinReflectPresent()) {
    if(kotlinDetector.isSuspendingFunction(method)) {
      return invokeSuspendingFunction(method, getBean(), args)
    }
  ...
```

전파되지 않는 thread local

```
public static Publisher<?> invokeSuspendingFunction( ... ) {
  invokeSuspendingFunction(Dispatchers.getUnconfined(), ... );
}
```

CoroutineUtils.java

```
abstract class CoWebFilter : WebFilter {
  final override fun filter( ... ): Mono<Void> {
    return mono(Dispatchers.Unconfined) {
      ...
```

CoWebFilter.kt

* 참고 : 현재는 CoWebFilter 를 커스텀 하여 넣어줄 수 있음Ref)

Ref) https://github.com/spring-projects/spring-framework/commit/d75a7c38180b217b19bfa890bc4fd4e63f4416b0

Coroutine 을 도입해보자 - (2) suspend Out-Port

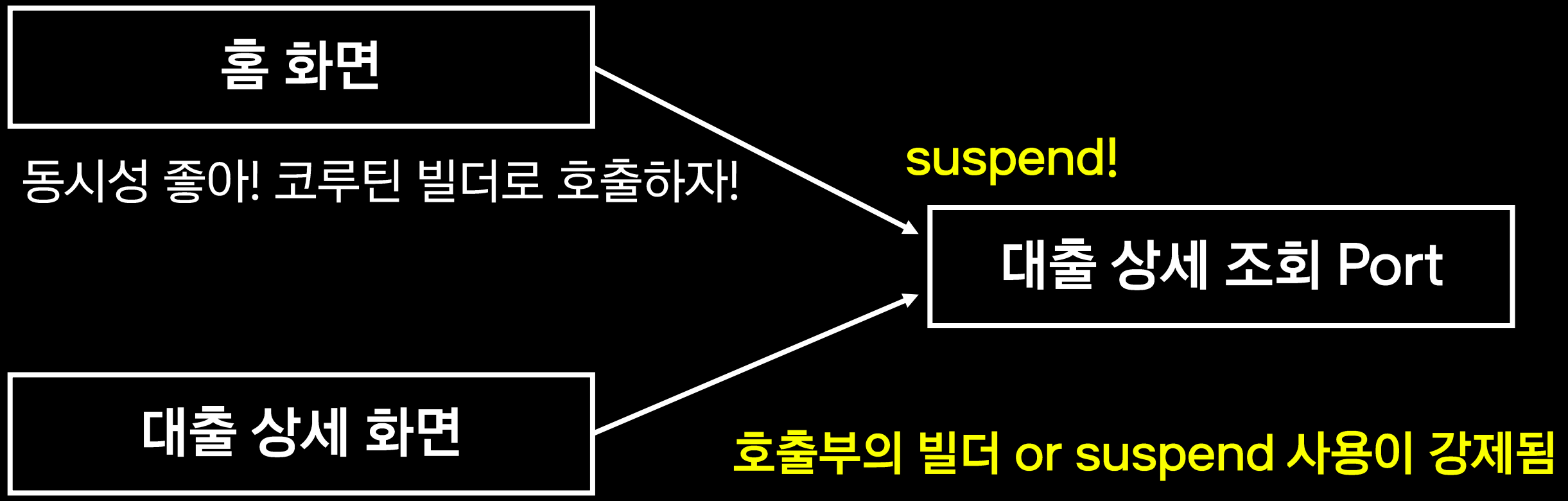
```
interface Outport {
  suspend fun 대출_정보_조회(): 대출_정보
  suspend fun 오픈뱅킹_정보_조회(): 오픈뱅킹_정보
```

```
fun 외부_서비스_조회( ... ): 외부_정보 {
  return runBlocking() {
    val 대출_정보 = 대출_정보_조회()
    val 오픈뱅킹_정보 = 오픈뱅킹_정보_조회()

    외부_정보(대출_정보, 오픈뱅킹_정보)
  }
```

- 컨텍스트 알아서 관리
- 간결한 코드
- 쓰레드 효율적 활용

전파되는 suspend



Coroutine 을 도입해보자 - (3) async/await 패턴

```
fun 외부_서비스_조회( ... ) {
  return runBlocking(코루틴_컨텍스트) {
    val 대출_정보 = async { 대출_정보_조회() }
    val 오픈뱅킹_정보 = async { 오픈뱅킹_정보_조회() }

    외부_정보(대출_정보.await(), 오픈뱅킹_정보.await())
  }
```

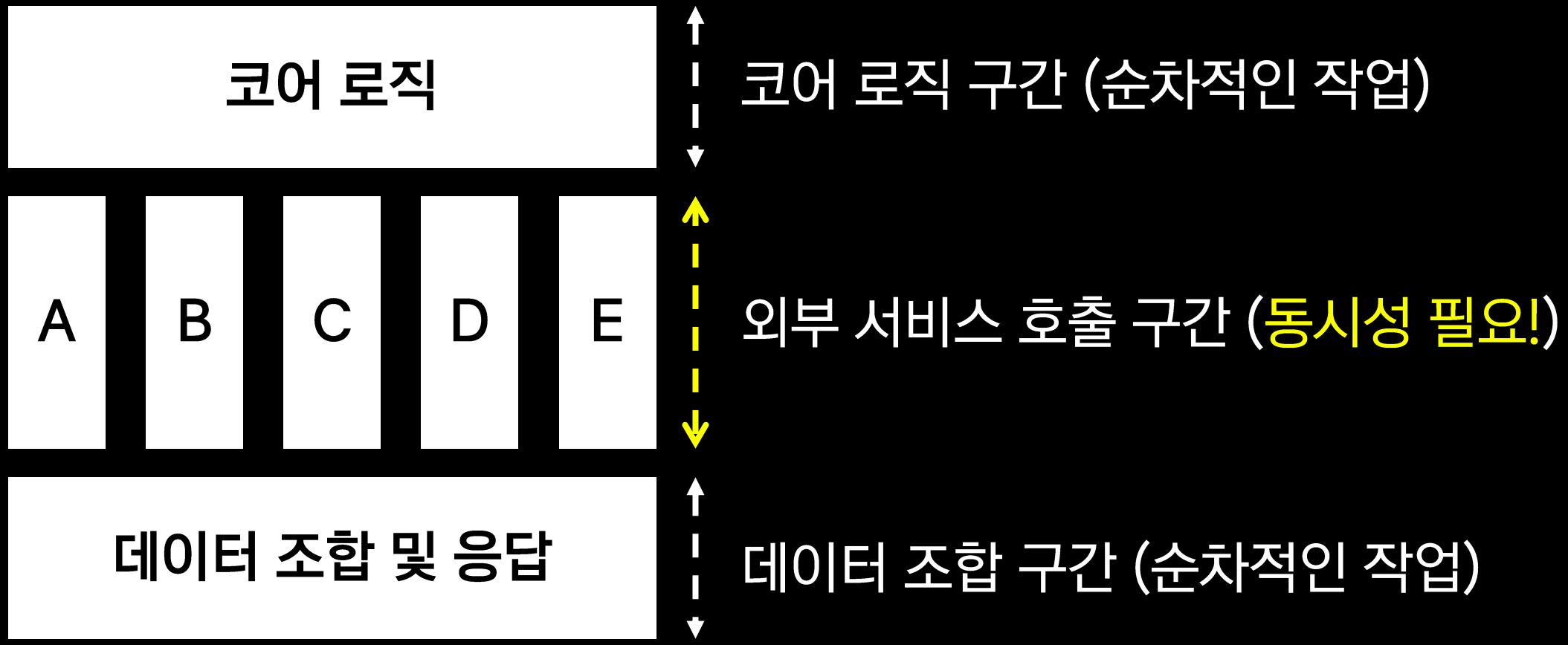
- 컨텍스트를 관리해야 함
- 중단함수를 사용하지 않기에 쓰레드 활용이 다소 비효율적임
- 동시성이 다른 곳으로 전파되지 않음

suspend 선택지의 트레이드 오프

- 별도의 코루틴 스코프, 컨텍스트 관리가 필요 없음
  (runBlocking 실행 시 컨텍스트 따로 안 넘겨줘도 됨)
- 코루틴을 더 효율적으로 사용할 수 있음

- 상위의 비동기 선택지가 하위 호출부까지 오염
  - 비동기 요구가 사라질 때 호출부가 함께 변경되어야 함 (SRP)
- 같은 메서드를 다른 곳에서 사용하는 경우, 코루틴 빌더 사용이 강제됨

우리에게 필요한 건...


코어 로직
A
B
C
D
E
데이터 조합 및 응답
코어 로직 구간 (순차적인 작업)
외부 서비스 호출 구간 (동시성 필요!)
데이터 조합 구간 (순차적인 작업)

Coroutine with async/await

- 우리는 모든 곳에서 동시성이 필요한 게 아님!
  필요한 곳에서만 사용하고 싶다!

- 성능을 비교해보니 suspend 의 성능이 조금 더 효율적이지만,
  크게 와닿을만한 수준의 차이는 아니었음

- 불필요한 동시성의 전파가 향후 더 큰 비용이 될 거라고 생각

선택은 트레이드 오프

"Architecture is the stuff you can't Google."

"There are no right or wrong answer in architecture, only trade-offs."

Ref) Mark Richards, Neal Ford, Fundamentals of Software Architecture

배경

코드 문제 해결

안정적 이관 전략

결과

장애가 발생해도 되는 시스템은 없지만..

- 돈을 다루기에 장애 발생 시 영향도가 큼
- 은행은 신뢰가 무엇보다 중요
- 인터넷 은행은 로컬 지점이 없기에 안정성이 더욱 중요

은행에서_장애_낸_개발자_끌려가는_영상.mp4

금융 감독원 장애 보고

- 핵심 업무 장애인 경우
- 금액, 횟수, 사용자 수 등 영향도가 큰 경우

홈 화면 장애 = 조회/이체 불가 = 핵심 업무 + 높은 영향도!

절대 장애가 발생하면 안 돼

Ref) SBS, 정글의 법칙

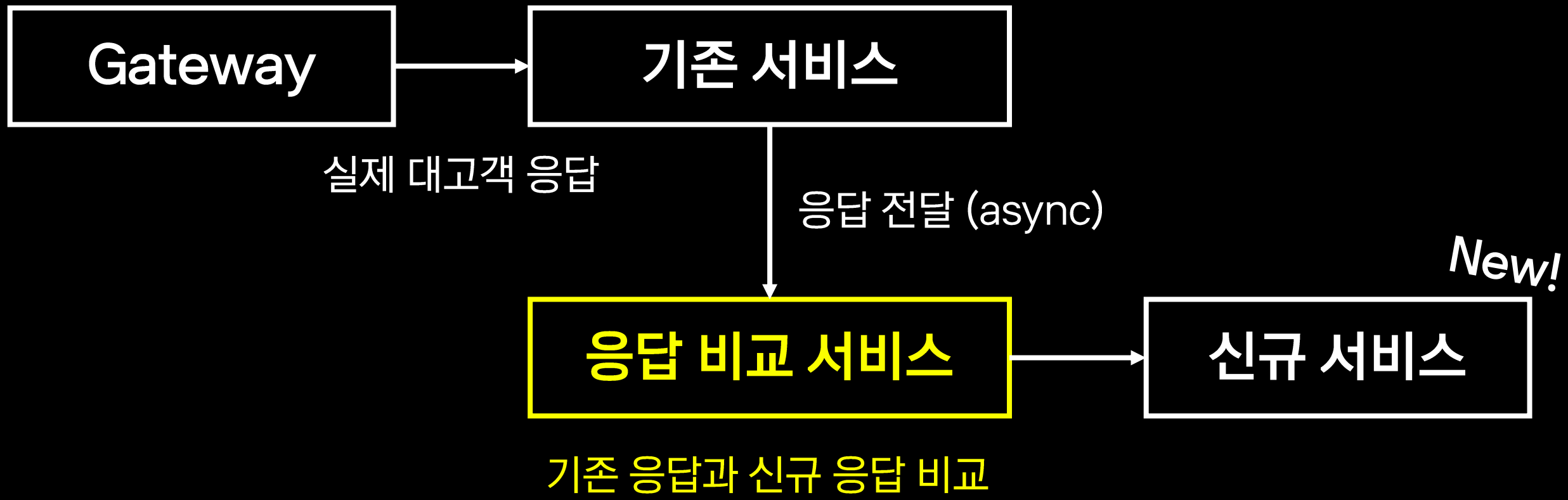
문제 1: 너무 커져버린 변경

- 기존 코드는 Java, 레이어 아키텍처, 동기식 호출
- 새로운 코드는 Kotlin, 헥사고날 아키텍처, 비동기 호출

- 사실상 아예 새롭게 만들어진 서비스

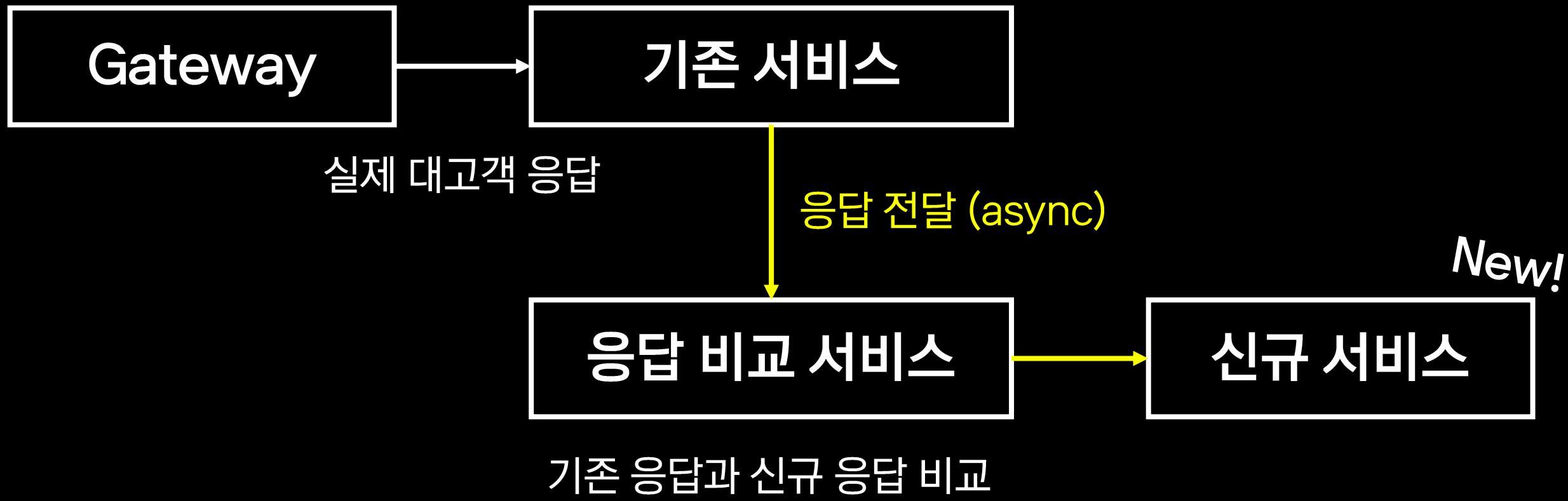
- 서버 만의 분리 작업이기에 외부 인터페이스가 변하면 안 됨!

안정성을 확보해보자 1 : 응답 비교

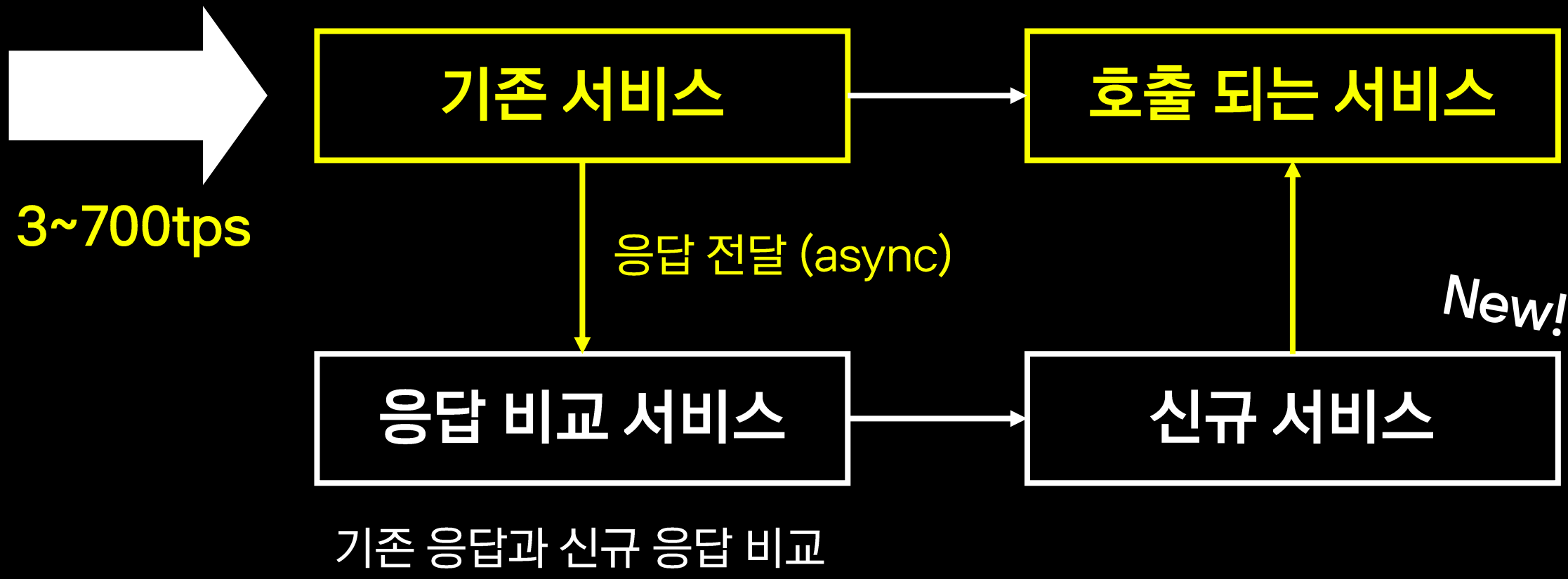


이슈: 시간차 공격

- 시간 차이에 의한 불일치 발생



문제 2: 비교 만으로도 영향 받는 시스템들

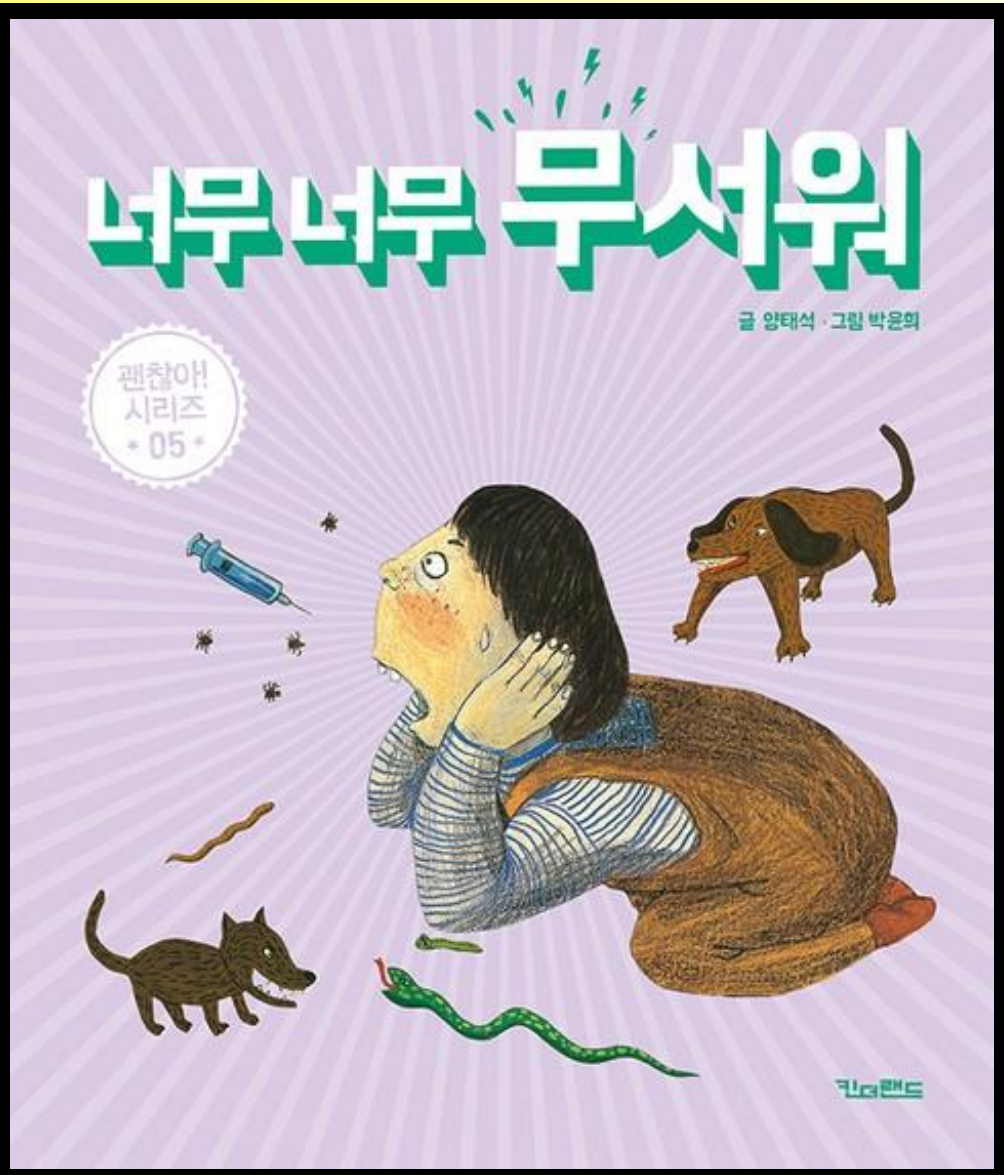


안정성을 확보해보자 2 : 표본검사

- 하루 약 3천만 호출
- 1% = 하루 30만 호출
- 1% 만 해도 상당히 많은 수의 요청 유형을 비교/검토해볼 수 있음

- 검증 비율을 동적으로 설정할 수 있도록 설계
- 0.1% 로 시작해서 점진적으로 검증(응답 비교) 비율을 확대
- 검증하며 각종 지표를 모니터링

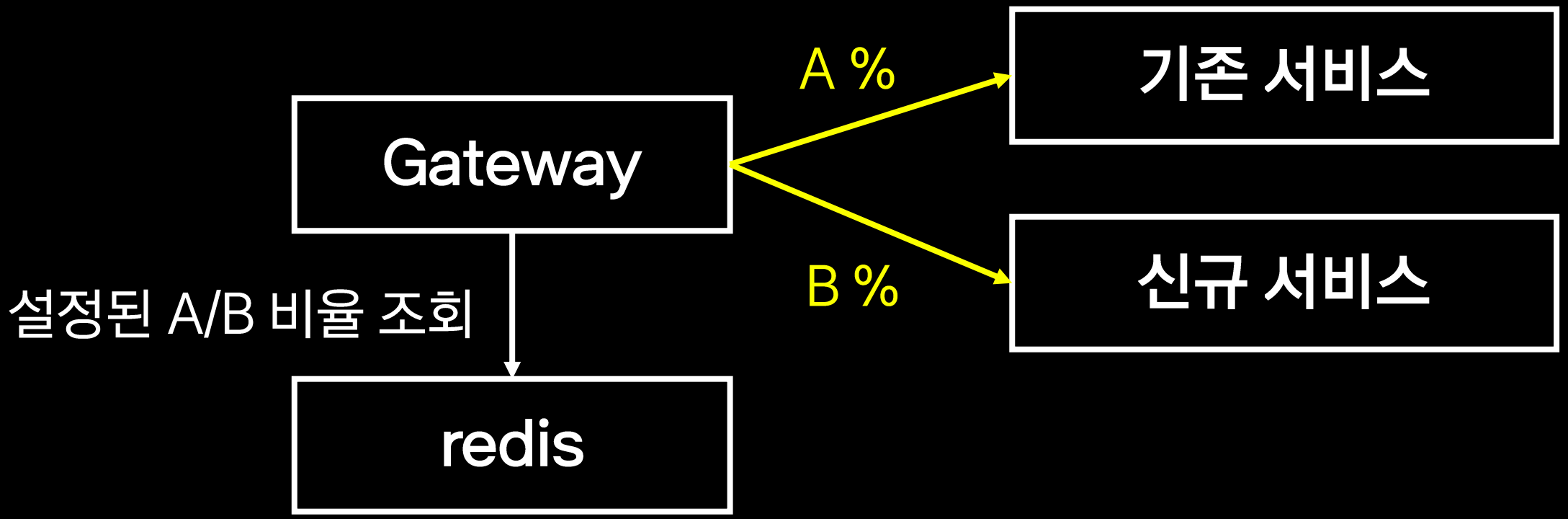
너무 너무 무서워



Ref) 양태석, 너무 너무 무서워

안정성을 확보해보자 3 : A/B

- A/B 테스트 하듯이 트래픽을 일정 비율로 전환
- 언제든지 비율이 변경될 수 있도록 동적으로 설정



이슈: 하이럼의 법칙

"With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviours of your system will be depended on by somebody."

Ref) Titus Winters and others, Software Engineering at Google

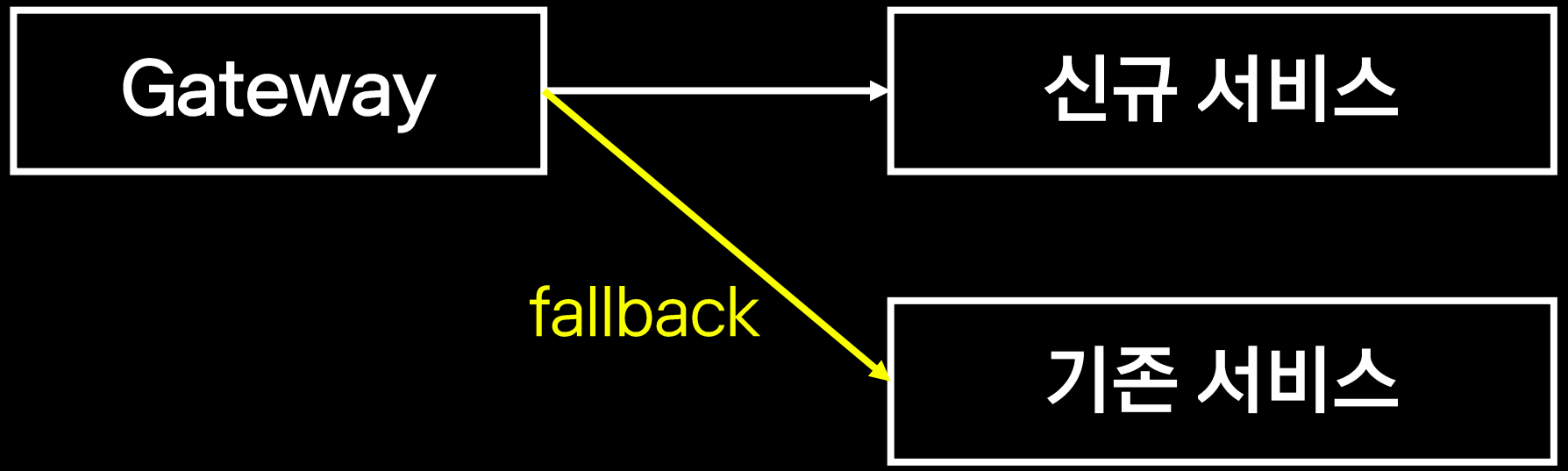
하이럼의 법칙 : 별칭 보정

- 별칭이 존재하지 않으면 보정해주는 기능이 있었음
- 신규 서비스에서는 이 기능을 제거
  - 홈 화면의 역할과 책임에 맞지 않다고 판단
  - 만약의 상황을 위해 존재했던 기능
  - 최근 호출되지 않은 것을 확인

- 나중에 다른 시스템에서 이 별칭 보정 기능을 의존하고 있었다는 걸 알게 됨

안정성을 확보해보자 4 : fallback

- 신규 서비스에서 예외가 발생하면 기존 서비스 응답으로 fallback



이슈: nginx 다운!

Ref) SBS, 그것이 알고 싶다

배경

코드 문제 해결

안정적 이관 전략

결과

정리 : 기술부채 해결

1) 외부 의존성과 도메인 정책이 혼재되어 섞여 있음

- Hexagonal Architecture 도입

2) 외부 서비스 호출 증가에 따른 성능상 우려

- Kotlin Coroutine 도입

훨씬 편해진 유지보수

Ref) 개비스콘 광고, 옥시레킷벤키저

기술 회고

- Hexagonal Architecture
  - 깃 충돌 거의 사라지고 협업이 더 쉬워짐
  - 간단한 기능 신규 구현할 때는 불편
- Kotlin Coroutine
  - (약간)빨라진 응답
  - 간결한 코드!

정리 : 안정적 이관 전략

- 응답의 일관성 유지
  - 응답 비교 서비스
- 다른 서비스에 대한 영향도 최소화
  - 표본 검사
- 장애 최소화 및 빠른 복구
  - A/B 트래픽
  - fallback

아무 일도.. 없었다..

Ref) 오다 에이치로, 원피스, 대원 씨아이

홈서비스야, 다치지 말고 오래 오래 행복하게 살아야 한다

Ref) TV동물농장, SBS

Q & A