Python -> Go Migration으로 TPS 30배 개선하기

신영민

AB180 Backend Engineer

Google Developer Groups

Cloud • Golang korea

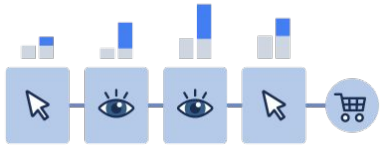
1. Migration 배경
2. 목적 및 방향성
3. 진행 과정
4. 결과

Google Developer Groups

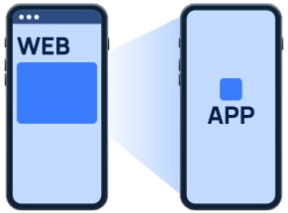
Migration
배경

Airbridge & Redirector

Airbridge:
People-Based Attribution and Incrementality Measurement for Web and Mobile.



유저 행동 분석



Web to App 광고 성과 분석

Airbridge & Redirector

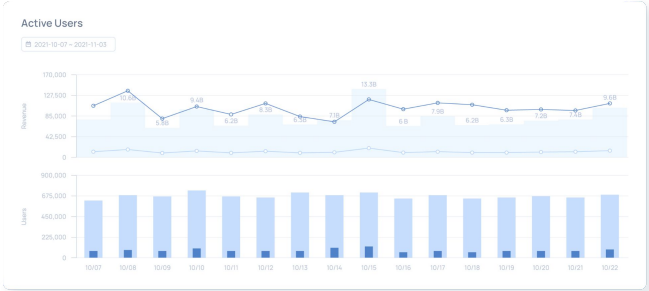
여러 지면에서 광고 링크를 통해 앱으로 연결

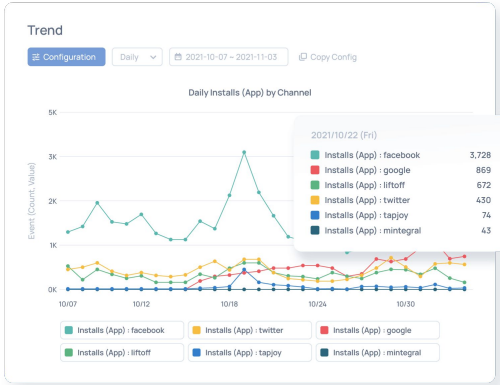
'image: Flaticon.com'

Airbridge & Redirector

<Redirector>

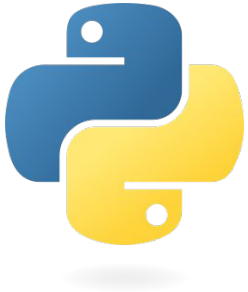
1) Daily 10억건의 광고 정보 수집
2) 목적지까지 Redirect



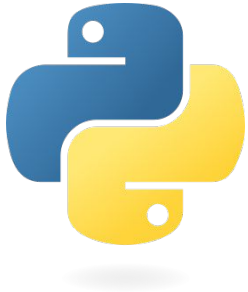


목적 및 방향성

Redirector 변천사



Python 2
Flask



Python 3
Sanic(asyncio)

What's Next?

'image: Flaticon.com'

언어 선정 과정

성능

비용 절감과 직결

개발 생산성, 트러블슈팅 난이도

주요 컴포넌트이며

복잡한 비즈니스 로직을 가지고 계속 기능이 추가/수정됨

언어에 대한 팀 숙련도

Learning Curve 및 Migration 기간 축소와 관련

HR

미래를 고려해야 하므로

언어 선정 과정

Golang
타 컴포넌트에서 이미 사용중

성능 python 보다 좋은 것은 검증됨

난이도 큰 걱정 없음

숙련도 후보 중 가장 높음

HR 괜찮다

Rust
도전적인 선택

성능 뛰어나다고 알려져 있음

난이도 어렵다고 알려져 있음

숙련도 거의 없음

HR 팀 내/외부 모두 쉽지 않을것으로 예상

Kotlin
대세를 따르는 선택

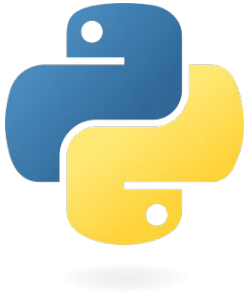
성능 python 보다는 좋을 것으로 기대

난이도 괜찮다고 알려져 있음

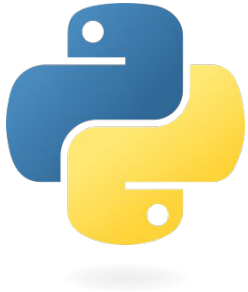
숙련도 Rust < Kotlin < Go

HR 가장 수월할 것으로 예상

PoC 결과



Python 2
Flask



Python 3
Sanic(asyncio)

Go!

Migration 방향성

- PoC
  - TPS, Error Rate 로 비교
  - General Case, Worst Case 각각을 비교
  - 외부 의존성을 포함한 시나리오로 검증
- 기능별(점진적) Migration 이 아니라 전면 교체
  - 비즈니스로직 특성에 맞는 전략 선택
- 리팩토링은 아래의 경우가 아니라면 최대한 지양
  - 중요한 비즈니스 로직 테스트 코드 작성 과정에서 어려움이 있을 때
  - 기타 언어적 한계로 인해 리팩토링이 필수적인 경우

진행 과정

사용한 Go Framework/Library

- Web Framework : Fiber + Jet6(template)
  - 기존 컴포넌트들은 Gin 사용중, 성능을 위해 fasthttp 기반의 Fiber 선택
  - Jinja와 비슷하면서 기존에 사용하던 기능들을 지원하는 Jet6 선택
- Infra : Gorm, Confluent-kafka-go, go-redis
  - 기존에 사용하고 있어서 그대로 선택

사용한 Go Framework/Library

- Configuration : viper, pflag, cobra
  - 기존에 사용하고 있어서 그대로 선택
  - viper, pflag 로 Nested Struct Path 명과 환경변수(or flag)로 주입할 수 있게 함
  - 그러나 boilerplate 가 많고 직관적이지 않아서 caarlos0/env 를 고려중
- DI : google/wire
  - 기존에 사용하고 있어서 그대로 선택
  - 오픈소스 관리가 안되고 있고, 에러 메시지가 직관적이지 않다는 의견이 있어 uber-go/fx 를 고려중

Migration 검증 과정

- End To End Tester 개발
  - 시나리오 생성
    - RDB, URL 등 Input을 Capture
    - Kafka 등 Output을 Capture
    - JSON 형태의 파일로 저장
  - 시나리오 검증
    - Input Context Setup
    - 비즈니스로직 실행
    - Output 비교


```
{
  "context": {
    "rdb": [
      {
        "table": "users",
        "row": {
          "id": 1,
          "name": "youngmin",
          ...
        }
      },
      ...
    ]
  },
  "request": {
    "url": "https://abit.ly/luft",
    "user_agent": "...",
    ...
  },
  "result": {
    "responses": [
      {
        "status_code": ...,
        "body": "...",
        "event": {...},
        ...
      }
    ],
    ...
  }
}
```

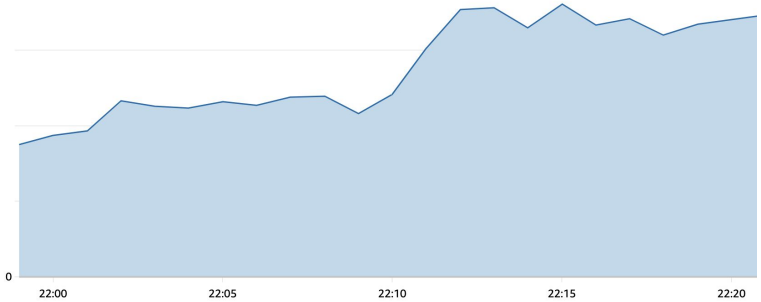

결과

Migration 결과

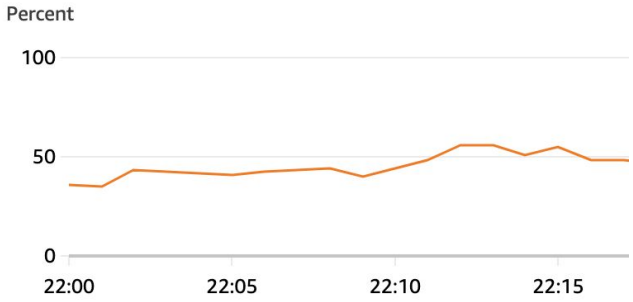
[Python]
Request Count

[Python]
Task Count

[Go]
Request Count



[Go]
CPU Utilization
(*No Change in Task Count)



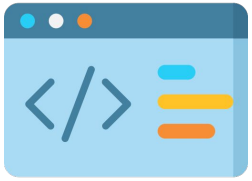
Migration 결과

성능

TPS 기준 **30**배 향상
Traffic Spike 에도
안정적

비용

약 **⅛** 로
감소



기타

강타입 언어..

'image: Flaticon.com'

Lessons Learned

- PoC 컴포넌트 개발 시 외부 의존성에 대한 설정들을 꼼꼼히 확인하자
- 오래된 컴포넌트일수록 Deprecate 시킬 수 있는 로직이 있는지 미리 확인하자
- 리팩토링은 최소화 하는것이 좋다
- 코드 작업보다 Diff 검증이 더 힘들었다

What's Next

- 기술 부채 해결
  - 복잡한 비즈니스로직을 DDD를 통해 분리하는 중
    - 다음엔 점진적 Migration이 가능한 상태를 만드는 것이 목표
  - 미처 옮기지 못한 Unit Test 보완
- 더 나은 Library 리서치(ex. wire -> uber-go/fx)
- 남은 다른 Python 컴포넌트도 Migration 고려

Q&A

AB180 Backend Engineer 신영민
Email : youngmin@ab180.co

함께 문제를 고민하고,
해결할 분들을 모십니다 :)
https://abit.ly/2023ab180

Google Developer Groups