캡스톤디자인 프로젝트 적용

S3 + CloudFront, Redis 캐싱, Async

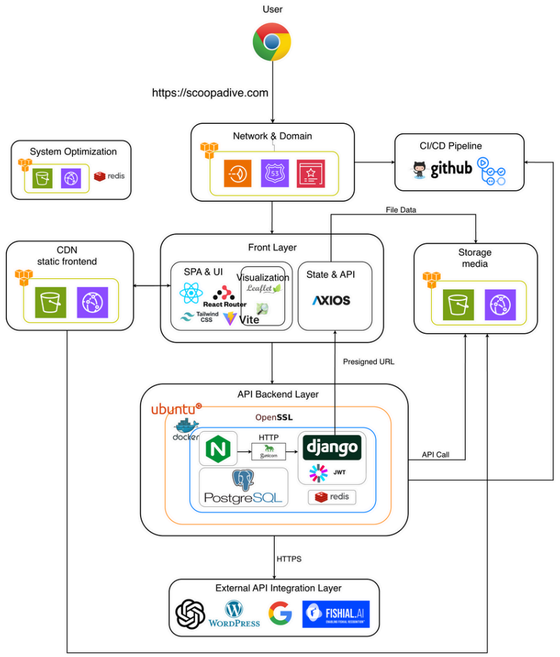
웹서비스 성능 최적화

21101223 전효정

목차

“프로젝트 선정동기”

01 프로젝트 선정동기



캡스톤디자인 프로젝트 시스템 아키텍처

빅데이터 처리 과목에서 배운 내용들
을 적용해 볼 수 있지 않을까?

“S3 + CloudFront I/O 부하 분산”

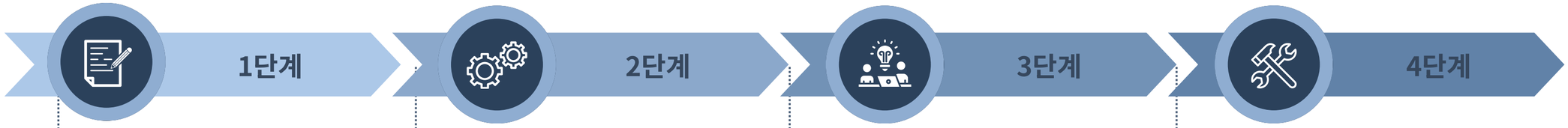
02 S3 + CloudFront I/O 부하 분산

정적 파일을 분리했을 때 성능이 향상된다?

기존 서버 파일 구조

하나의 Lightsail 인스턴스에 Backend (API 요청), Frontend (정적 파일) 두 가지 폴더가 모두 있다.

02 S3 + CloudFront I/O 부하 분산 프로세스



S3 버킷 생성

- 정적 웹사이트 호스팅 활성화

CloudFront 배포 생성

- Origin: S3 버킷

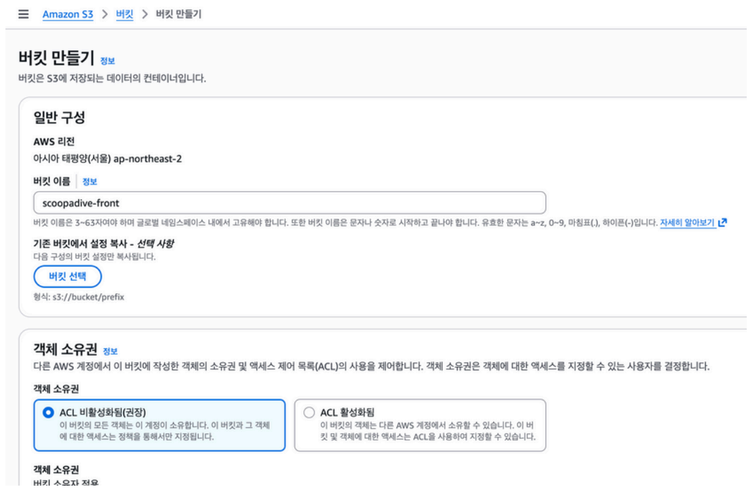
Front 빌드 파일 업로드

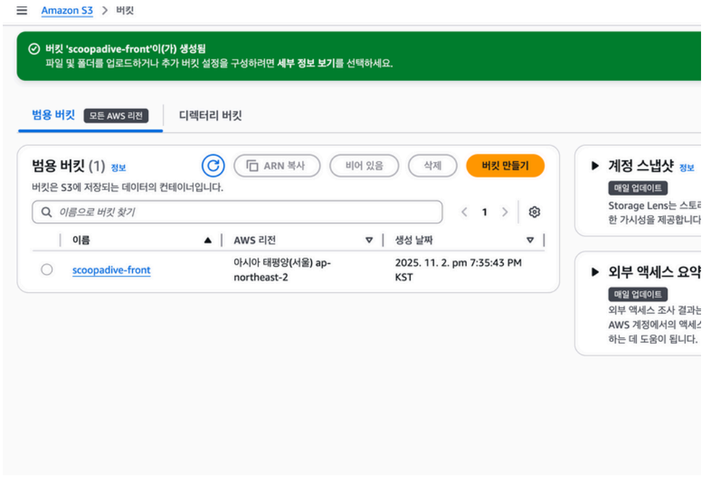
- npm run build 후 /dist 폴더 내용 S3 업로드

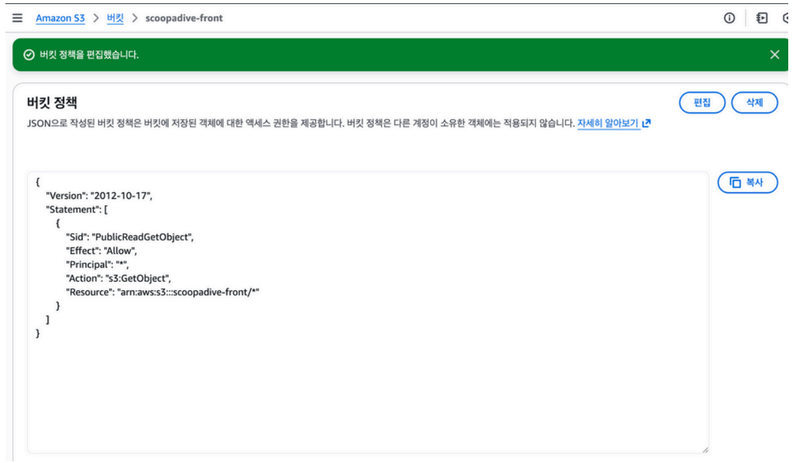
Nginx 서버 경로 제거 및 도메인 연결

- Nginx는 Backend API 만 처리
- CloudFront → Route53

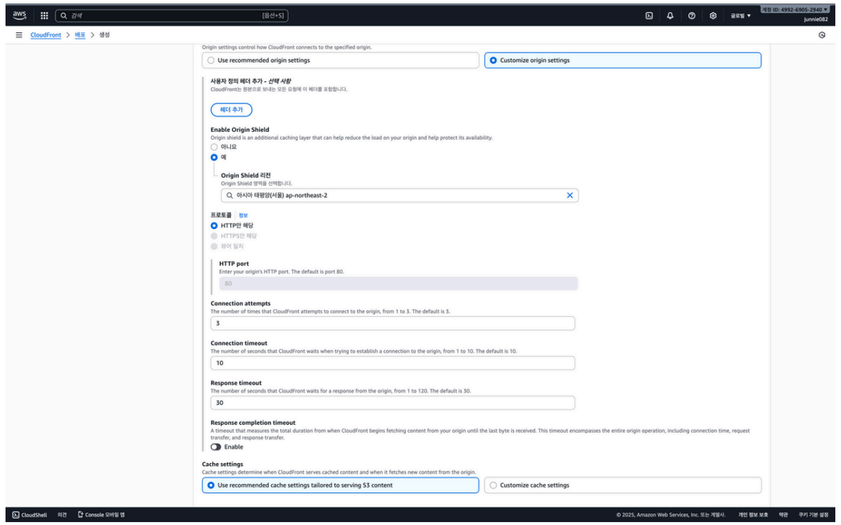
S3 버킷 생성

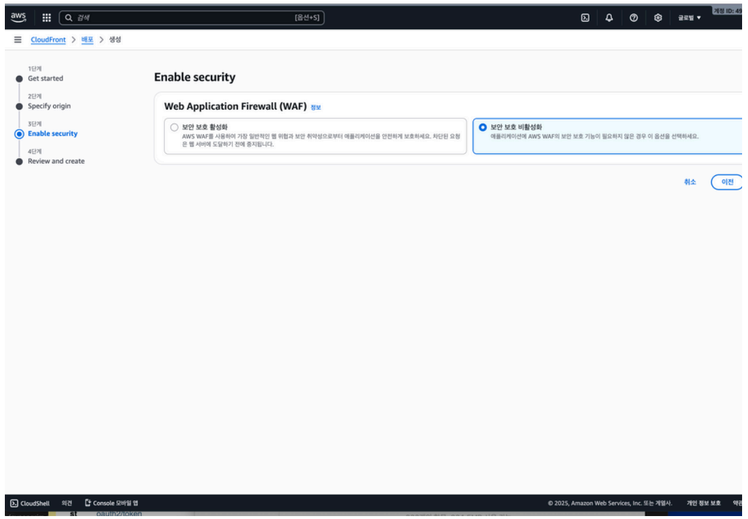

Amazon S3 > 버킷 > 버킷 만들기
버킷 만들기 정보
버킷은 S3에 저장되는 데이터의 컨테이너입니다.
일반 구성
AWS 리전
아시아 태평양(서울) ap-northeast-2
버킷 이름 정보
scoopadive-front
버킷 이름은 3~63자이야 하며 글로벌 네임스페이스 내에서 고유해야 합니다. 또한 버킷 이름은 문자나 숫자로 시작하고 끝나야 합니다. 유효한 문자는 a~z, 0~9, 마침표(.), 하이픈(-)입니다. 자세히 알아보기
기존 버킷에서 설정 복사 - 선택 사항
다음 구성의 버킷 설정만 복사됩니다.
버킷 선택
형식: s3://bucket/prefix
객체 소유권 정보
다른 AWS 계정에서 이 버킷에 작성한 객체의 소유권 및 액세스 제어 목록(ACL)의 사용을 제어합니다. 객체 소유권은 객체에 대한 액세스를 지정할 수 있는 사용자를 결정합니다.
객체 소유권
ACL 비활성화됨(권장)
이 버킷의 모든 객체는 이 계정이 소유합니다. 이 버킷과 그 객체에 대한 액세스는 정책을 통해서만 지정됩니다.
ACL 활성화됨
이 버킷의 객체는 다른 AWS 계정에서 소유할 수 있습니다. 이 버킷 및 객체에 대한 액세스는 ACL을 사용하여 지정할 수 있습니다.
객체 소유권

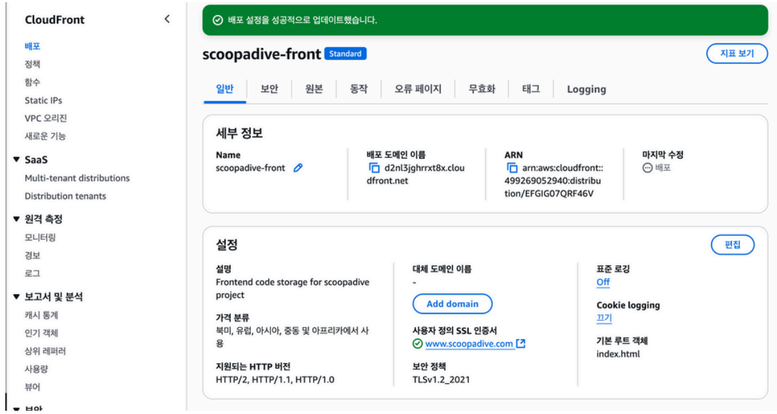

Amazon S3 > 버킷
버킷 'scoopadive-front'이(가) 생성됨
파일 및 폴더를 업로드하거나 추가 버킷 설정을 구성하려면 세부 정보 보기를 선택하세요.
범용 버킷
모든 AWS 리전
디렉터리 버킷
범용 버킷 (1) 정보
버킷은 S3에 저장되는 데이터의 컨테이너입니다.
ARN 복사
비어 있음
삭제
버킷 만들기
이름으로 버킷 찾기
이름
AWS 리전
생성 날짜
scoopadive-front
아시아 태평양(서울) ap-northeast-2
2025. 11. 2. pm 7:35:43 PM KST
계정 스냅샷 정보
매일 업데이트
외부 액세스 요약
매일 업데이트


Amazon S3 > 버킷 > scoopadive-front
버킷 정책을 편집했습니다.
버킷 정책
JSON으로 작성된 버킷 정책은 버킷에 저장된 객체에 대한 액세스 권한을 제공합니다. 버킷 정책은 다른 계정이 소유한 객체에는 적용되지 않습니다. 자세히 알아보기
편집
삭제
복사
{
"Version": "2012-10-17",
"Statement": [
{
"Sid": "PublicReadGetObject",
"Effect": "Allow",
"Principal": "*",
"Action": "s3:GetObject",
"Resource": "arn:aws:s3:::scoopadive-front/*"
}
]
}

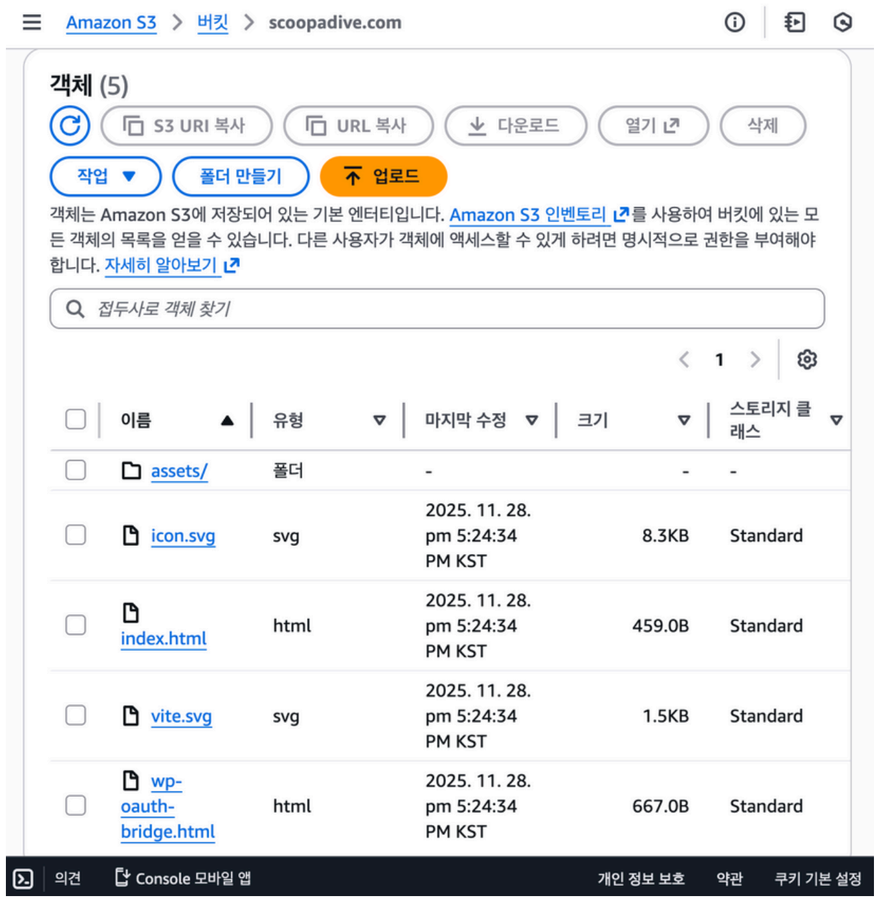
CloudFront 배포 생성







Front 빌드 파일 업로드



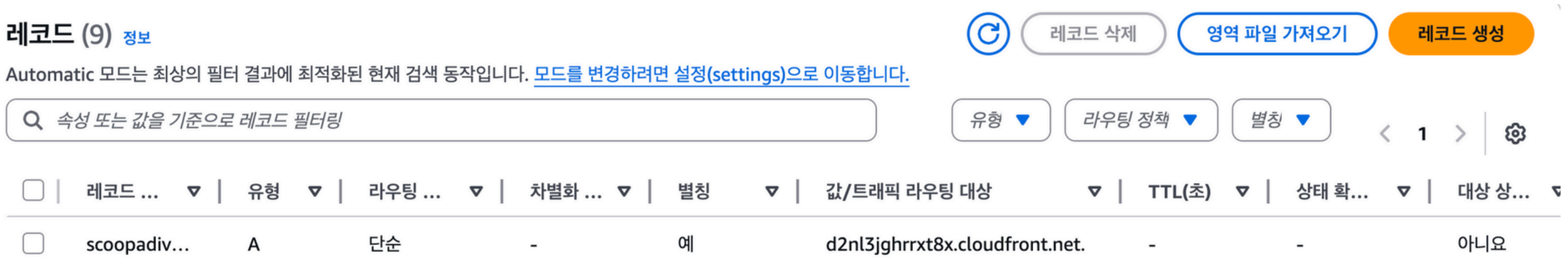
npm run build
aws s3 sync dist/ s3://scoopadive-front/ --delete

Nginx 경로 제거 및 도메인 연결

```
nginx:
  image: nginx:latest
  container_name: nginx_proxy
  ports:
    - "80:80"
    - "443:443"
  volumes:
    - /etc/letsencrypt:/etc/letsencrypt:ro
    - /home/ubuntu/Backend/nginx.conf:/etc/nginx/conf.d/defau
    - /home/ubuntu/Frontend/dist:/usr/share/nginx/html:ro
    - static_volume:/scoopadive/staticfiles
```

/home/ubuntu/Frontend 경로 삭제

Nginx 경로 제거 및 도메인 연결



CloudFront → Route53 도메인 연결

k6 부하테스트

K6 부하테스트란?

- Grafana Labs 에서 만든 오픈소스 성능 테스트 도구인 k6를 사용하여 웹 애플리케이션이나 API 에 부하를 발생시키고 성능을 측정하는 것.

- http_reqs: 총 얼마나 많은 HTTP requests를 k6 에서 생성했는지
- http_req_duration: 요청의 총 시간

Lightsail 직접 서빙


```
import http from 'k6/http';
import { sleep } from 'k6';

export let options = {
          vus: 100,
          duration: '2m',
};

export default function () {
          http.get('https://scoopadive.com'); // Lightsail
          sleep(0.5);
}

~
```

http_req_duration... avg=41.44ms

CloudFront 분리 후

```
❯ k6 run cloudfront_test.js

         /\      Grafana   /‾‾/
    /\  /  \     |\  __   /  /
   /  \/    \    | |/ /  /   ‾‾\
  /          \   |   (  |  (‾)  |
 / __________ \  |_|\_\  \_____/

     execution: local
        script: cloudfront_test.js
        output: -

     scenarios: (100.00%) 1 scenario, 100 max VUs, 2m30s max duration (incl. graceful stop):
              * default: 100 looping VUs for 2m0s (gracefulStop: 30s)


  TOTAL RESULTS

    HTTP
    http_req_duration..............: avg=14.85ms  min=6.58ms   med=11.65ms  max=253.59ms p(90)=20.12ms  p(95)=31.72ms
      { expected_response:true }...: avg=14.85ms  min=6.58ms   med=11.65ms  max=253.59ms p(90)=20.12ms  p(95)=31.72ms
    http_req_failed................: 0.00%  0 out of 23316
    http_reqs......................: 23316  193.614543/s

    EXECUTION
    iteration_duration.............: avg=516.09ms min=507.48ms med=512.41ms max=754.95ms p(90)=521.62ms p(95)=534.81ms
    iterations.....................: 23316  193.614543/s
    vus............................: 100    min=100        max=100
    vus_max........................: 100    min=100        max=100

    NETWORK
    data_received..................: 17 MB  142 kB/s
    data_sent......................: 2.4 MB 20 kB/s



running (2m00.4s), 000/100 VUs, 23316 complete and 0 interrupted iterations
default ✓ [======================================] 100 VUs  2m0s
❯ cat cloudfront_test.js
import http from 'k6/http';
import { sleep } from 'k6';

export let options = {
          vus: 100,
          duration: '2m',
};

export default function () {
          http.get('https://api.scoopadive.com'); // CloudFront에서 프론트 서빙 중인 주소
          sleep(0.5);
}
```

```
import http from 'k6/http';
import { sleep } from 'k6';

export let options = {
          vus: 100,
          duration: '2m',
};

export default function () {
          http.get('https://api.scoopadive.com'); // CloudFront에서 프론트 서빙 중인 주소
          sleep(0.5);
}
```

http_req_duration... avg=14.85ms

실험 결과 요약

지표	Lightsail 직접 서빙	CloudFront 분리 후	개선률
avg 응답시간	41.44ms	14.85ms	↓ 64.2%
median (중앙값)	32.57ms	11.65ms	↓ 64.2%
P90 응답시간	60.25ms	20.12ms	↓ 66.6%
MAX 응답시간	539.83ms	253.59ms	↓ 53.0%
req/s	183.72s	193.61s	+5.3%

CloudFront 분리 후 성능

- 평균 응답시간이 41.44 → 14.85ms 로 64% 감소
- 90% 지연시간도 66.6% 감소
- 정적인 SPA 리소스가 캐시로 가면서 백엔드가 매우 가벼워짐
- 서버가 프론트를 서빙하지 않을 때 API 응답이 즉시적으로 빨라졌다는 것을 의미함

CloudFront 분리 후 처리량

- Before: 183.72 req/s
- After: 193.61 req/s
- 5.3% 증가
- 서버가 더 높은 양의 요청을 부담없이 처리할 수 있음을 보여줌

결론

정적 파일이 Lightsail → CloudFront 로 옮겨가면서 Lightsail은
실질적으로 API 전용 서버가 되었고,
그 결과 **성능 및 안정성 모두 크게 향상되었다**.

Redis 캐싱

03

Redis 캐싱

worker 수를 늘리면 성능이 향상된다?

```
command: >
  sh -c "
  python manage.py collectstatic --noinput &&
  python manage.py migrate &&
  gunicorn scoopadive.asgi:application -k uvicorn.workers.UvicornWorker --workers 4 --bind 0.0.0.0:8000
  "
```

더 많은 일꾼 = 더 많은 일 처리

03

Redis 캐싱

```
@action(detail=False, methods=['get'])
def likes(self, request):
    data = [
        {
            'id': logbook.id,
            'likes': list(logbook.likes.values_list('username', flat=True))
        }
        for logbook in self.queryset if logbook.likes.exists()
    ]
    return Response(data)
```

비동기화 시키면 성능이 향상된다?

기존에 '좋아요'를 읽어 들이는 API

루프 안에서 처리하는 작업은 느리므로 병렬적으로 처리하면 빨라질 것이다.

Async/Await는 여러 작업을 동시에 처리하게 해준다.

03

Redis 캐싱

```python
@action(detail=False, methods=['get'])
async def likes_async(self, request):
    from asgiref.sync import sync_to_async

    async_get_likes = sync_to_async(
        lambda: [
            {
                'id': logbook.id,
                'likes': list(logbook.likes.values_list('username', flat=True))
            }
            for logbook in Logbook.objects.all().prefetch_related("likes")
        ],
        thread_sensitive=True
    )

    data = await async_get_likes()
    return Response(data)
```

비동기화 시키면 성능이 향상된다?

기존에 '좋아요'를 읽어 들이는 API

루프 안에서 처리하는 작업은 느리므로 병렬적으로 처리하면 빨라질 것이다.

Async/Await는 여러 작업을 동시에 처리하게 해준다.

03 동기/비동기식 처리 및 Worker 수 증가 실험

sync + worker

동기식 처리에서 worker 수를 늘려가며 실험

async + worker

비동기식 처리에서 worker 수를 늘려가며 실험

sync vs async

sync 와 async 처리 비교

Redis 캐싱

문제점 발견 및 해결 방법

적절한 vus 값 찾기

```
command: >
  sh -c "
  python manage.py collectstatic --noinput &&
  python manage.py migrate &&
  gunicorn scoopadive.asgi:application -k uvicorn.workers.UvicornWorker --workers 2 --bind 0.0.0.0:8000
  "
```

```
import http from "k6/http";
import { check } from "k6";

const BASE = "https://scoopadive.com";

export let options = {
    vus: 10,
    duration: "30s",
};

export default function () {
  // Login
  const loginPayload = JSON.stringify({
    email: "ruby@gmail.com",
    password: "12345678"
  });

  const loginHeaders = { "Content-Type": "application/json" };
  const loginRes = http.post(`${BASE}/api/auths/signin/`, loginPayload, { headers: loginHeaders });

  check(loginRes, { "login succeeded": (r) => r.status === 200 });

  const accessToken = loginRes.json("access");

  // API
  const apiHeaders = {
    "Authorization": `Bearer ${accessToken}`,
  };

  const res = http.get(`${BASE}/api/logbooks/likes/`, { headers: apiHeaders });

  check(res, { "status was 200": (r) => r.status === 200 });
}
~
~
```

worker = 2

vus = 10

worker = 2, vus = 10

http_req_duration... avg=2.1ms

vus = 50

worker = 2, vus = 50

```
        /\      Grafana   /‾‾/
   /\  /  \     |\  __   /  /
  /  \/    \    | |/ /  /   ‾‾\
 /          \   |   (  |  (‾)  |
/ __________ \  |_|\_\  \_____/

     execution: local
        script: likes_sync.js
        output: -

     scenarios: (100.00%) 1 scenario, 50 max VUs, 1m0s max duration (incl. graceful stop):
              * default: 50 looping VUs for 30s (gracefulStop: 30s)


  TOTAL RESULTS

     checks_total.......: 200     4.728257/s
     checks_succeeded...: 100.00% 200 out of 200
     checks_failed......: 0.00%   0 out of 200

     ✓ login succeeded
     ✓ status was 200

     HTTP
     http_req_duration..............: avg=10.47s min=431.75ms med=10.46s max=20.73s p(90)=18.36s p(95
)=19.37s
       { expected_response:true }...: avg=10.47s min=431.75ms med=10.46s max=20.73s p(90)=18.36s p(95
)=19.37s
     http_req_failed................: 0.00%  0 out of 200
     http_reqs......................: 200    4.728257/s

     EXECUTION
     iteration_duration.............: avg=20.98s min=20.79s   med=20.83s max=21.49s p(90)=21.36s p(95
)=21.43s
     iterations.....................: 100    2.364128/s
     vus............................: 25     min=25       max=50
     vus_max........................: 50     min=50       max=50

     NETWORK
     data_received..................: 439 kB 10 kB/s
     data_sent......................: 126 kB 3.0 kB/s
```

http_req_duration... avg=10.47ms

vus = 100

worker = 2, vus = 100

```
        /\      Grafana   /‾‾/
     /\  /  \     |\  __   /  /
    /  \/    \    | |/ /  /   ‾‾\
   /          \   |   (  |  (‾)  |
  / __________ \  |_|\_\  \_____/

     execution: local
        script: likes_sync.js
        output: -

     scenarios: (100.00%) 1 scenario, 50 max VUs, 1m0s max duration (incl. graceful stop):
              * default: 50 looping VUs for 30s (gracefulStop: 30s)


  TOTAL RESULTS

    checks_total.......: 200     4.728257/s
    checks_succeeded...: 100.00% 200 out of 200
    checks_failed......: 0.00%   0 out of 200

    ✓ login succeeded
    ✓ status was 200

    HTTP
    http_req_duration..............: avg=10.47s min=431.75ms med=10.46s max=20.73s p(90)=18.36s p(95
)=19.37s
      { expected_response:true }...: avg=10.47s min=431.75ms med=10.46s max=20.73s p(90)=18.36s p(95
)=19.37s
    http_req_failed................: 0.00%  0 out of 200
    http_reqs......................: 200    4.728257/s

    EXECUTION
    iteration_duration.............: avg=20.98s min=20.79s   med=20.83s max=21.49s p(90)=21.36s p(95
)=21.43s
    iterations.....................: 100    2.364128/s
    vus............................: 25     min=25       max=50
    vus_max........................: 50     min=50       max=50

    NETWORK
    data_received..................: 439 kB 10 kB/s
    data_sent......................: 126 kB 3.0 kB/s
```

http_req_duration... avg=27.84ms

동기식 처리에서 worker 수를 늘리면

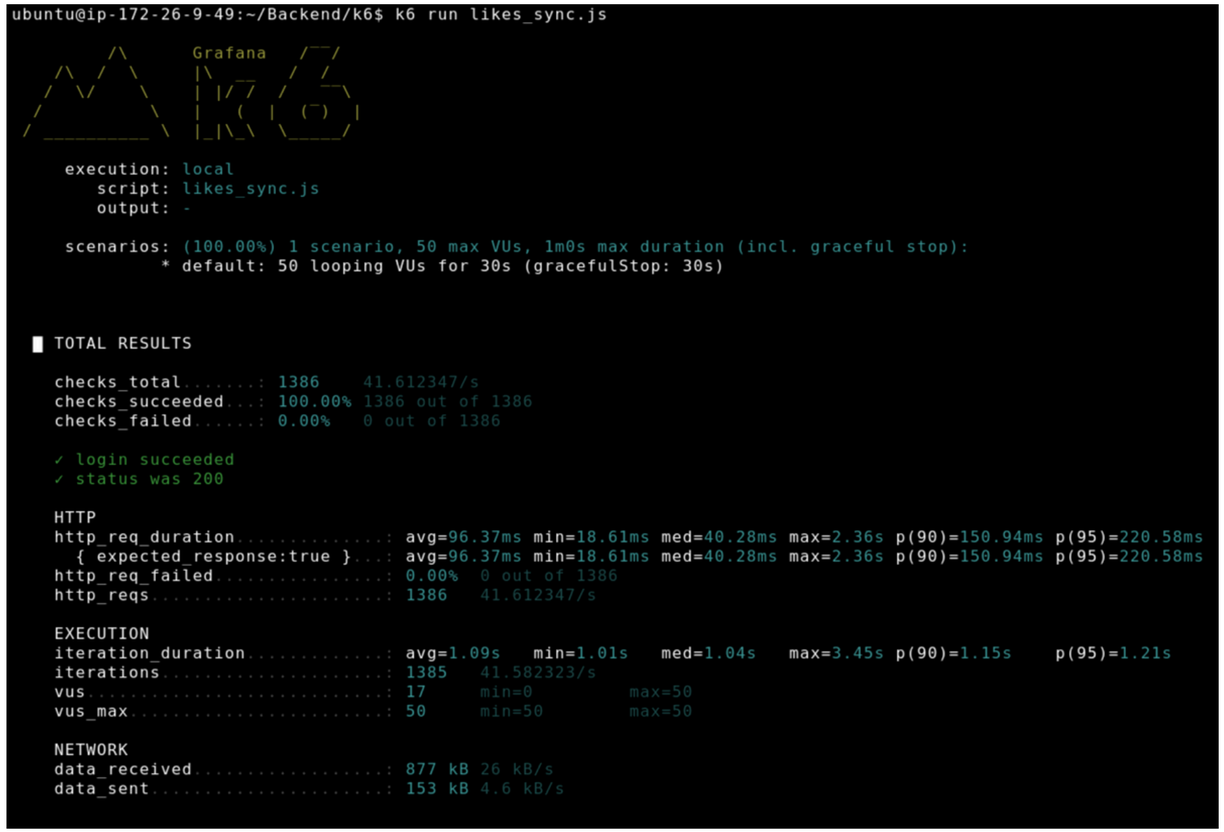
sync + worker = 1

worker = 1, vus = 50

http_req_duration... avg=100.58ms

sync + worker = 2

worker = 2, vus = 50

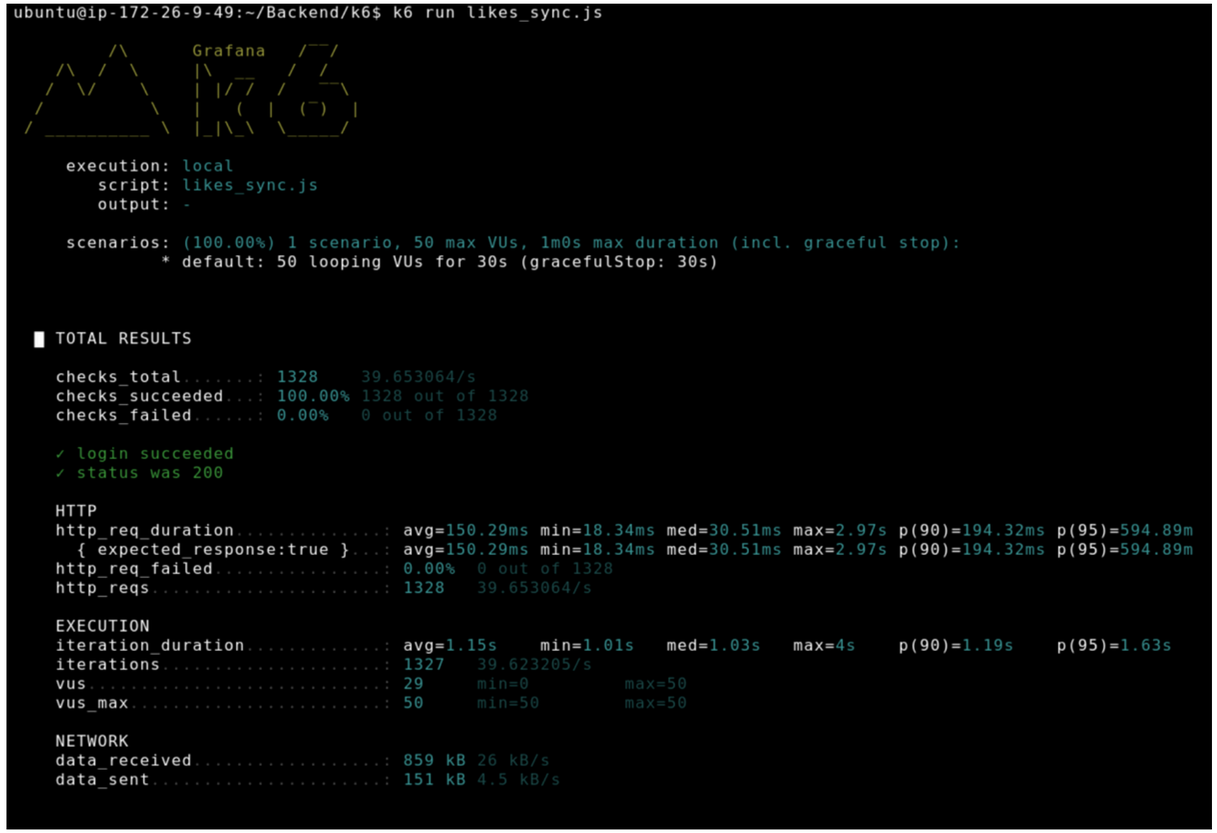


```
ubuntu@ip-172-26-9-49:~/Backend/k6$ k6 run likes_sync.js

         /\      Grafana   /‾‾/
    /\  /  \     |\  __   /  /
   /  \/    \    | |/ /  /   ‾‾\
  /          \   |   (  |  (‾)  |
 / __________ \  |_|\_\  \_____/

     execution: local
        script: likes_sync.js
        output: -

     scenarios: (100.00%) 1 scenario, 50 max VUs, 1m0s max duration (incl. graceful stop):
              * default: 50 looping VUs for 30s (gracefulStop: 30s)


  TOTAL RESULTS

    checks_total.......: 1386    41.612347/s
    checks_succeeded...: 100.00% 1386 out of 1386
    checks_failed......: 0.00%   0 out of 1386

    ✓ login succeeded
    ✓ status was 200

    HTTP
    http_req_duration..............: avg=96.37ms min=18.61ms med=40.28ms max=2.36s p(90)=150.94ms p(95)=220.58ms
      { expected_response:true }...: avg=96.37ms min=18.61ms med=40.28ms max=2.36s p(90)=150.94ms p(95)=220.58ms
    http_req_failed................: 0.00%  0 out of 1386
    http_reqs......................: 1386   41.612347/s

    EXECUTION
    iteration_duration.............: avg=1.09s   min=1.01s   med=1.04s   max=3.45s p(90)=1.15s    p(95)=1.21s
    iterations.....................: 1385   41.582323/s
    vus............................: 17     min=0         max=50
    vus_max........................: 50     min=50        max=50

    NETWORK
    data_received..................: 877 kB 26 kB/s
    data_sent......................: 153 kB 4.6 kB/s
```

http_req_duration... avg=96.37ms

sync + worker = 3

worker = 3, vus = 50



```
ubuntu@ip-172-26-9-49:~/Backend/k6$ k6 run likes_sync.js

         /\      Grafana   /‾‾/
    /\  /  \     |\  __   /  /
   /  \/    \    | |/ /  /   ‾‾\
  /          \   |   (  |  (‾)  |
 / __________ \  |_|\_\  \_____/

     execution: local
        script: likes_sync.js
        output: -

     scenarios: (100.00%) 1 scenario, 50 max VUs, 1m0s max duration (incl. graceful stop):
              * default: 50 looping VUs for 30s (gracefulStop: 30s)


  TOTAL RESULTS

    checks_total.......: 1328    39.653064/s
    checks_succeeded...: 100.00% 1328 out of 1328
    checks_failed......: 0.00%   0 out of 1328

    ✓ login succeeded
    ✓ status was 200

    HTTP
    http_req_duration..............: avg=150.29ms min=18.34ms med=30.51ms max=2.97s p(90)=194.32ms p(95)=594.89m
      { expected_response:true }...: avg=150.29ms min=18.34ms med=30.51ms max=2.97s p(90)=194.32ms p(95)=594.89m
    http_req_failed................: 0.00%  0 out of 1328
    http_reqs......................: 1328   39.653064/s

    EXECUTION
    iteration_duration.............: avg=1.15s    min=1.01s   med=1.03s   max=4s    p(90)=1.19s    p(95)=1.63s
    iterations.....................: 1327   39.623205/s
    vus............................: 29     min=0         max=50
    vus_max........................: 50     min=50        max=50

    NETWORK
    data_received..................: 859 kB 26 kB/s
    data_sent......................: 151 kB 4.5 kB/s
```

http_req_duration... avg=150.29ms

sync + worker = 4

worker = 4, vus = 50

http_req_duration... avg=152.9ms

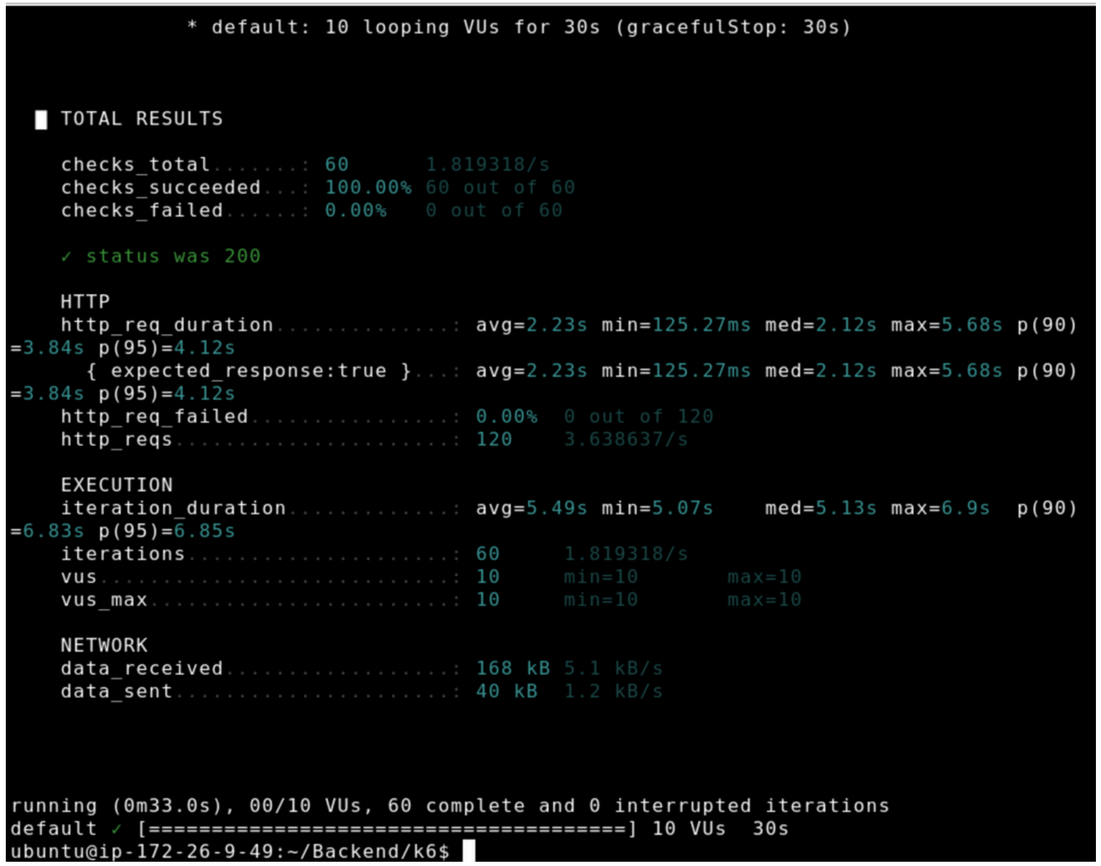
sync + worker 실험 관찰 결과

Baseline (sync)에서 worker 수를 늘리면
http_req_duration... avg 가 늘어난다...?

비동기식 처리에서 worker 수를 늘리면

async + worker = 1

worker = 1, vus = 50



http_req_duration... avg=87.61ms

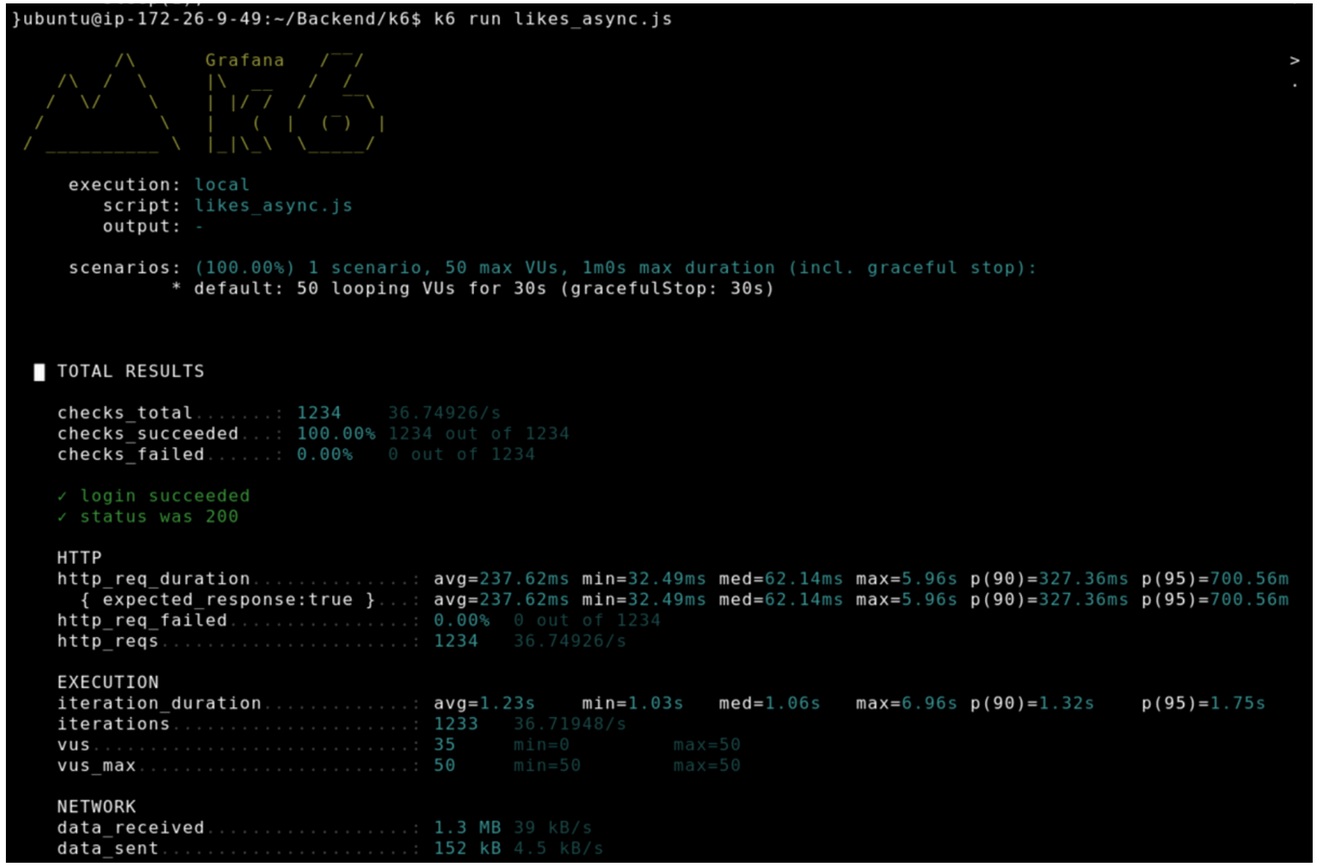
async + worker = 2

worker = 2, vus = 50

http_req_duration... avg=129.59ms

async + worker = 4

worker = 4, vus = 50



http_req_duration... avg=237.62ms

async + worker 실험 관찰 결과

비동기식 처리 (async)에서 worker 수를 늘리면
http_req_duration... avg 가 늘어난다...?

sync vs. async

sync vs. async 실험 결과

vus = 50 일 때 평균 응답 시간

종류/worker 수	1	2	4
sync (baseline)	100.58ms	96.37ms	152.9ms
async	87.61ms	129.59ms	237.62ms

동기/비동기식 처리 및 worker 수 증가 실험 결론 1

초기 가설: worker 수 를 늘리면 성능이 향상된다.

worker **2** → **4**개 전환시 성능 역효과:

- Sync: 96.37ms → 152.9ms (**58.6**% 지연)
- Async: 129.59ms → 237.62ms (**83.4**% 지연)

결론: worker 를 무작정 늘리는 것은 오히려 **치명적인 역효과를** 불러온다.

동기/비동기식 처리 및 worker 수 증가 실험 결론 2

초기 가설: 동기식 처리를 비동기식 처리로 바꾸면 성능이 향상된다.

- worker = 2일 때:

Sync: 96.37ms, Async: 129.59ms

- worker = 4일 때:

Sync: 152.9ms, Async: 237.62ms

결론: 비동기식으로 바꾸었을 때 오히려 **성능이 떨어지는 경우가** 있다.

원인 분석 1. N+1 쿼리 문제 & 지연로딩

반복문 안의 logbook.likes.exists() 때문에 Logbook 수만큼 DB 쿼리 발생

```
@action(detail=False, methods=['get'])
def likes(self, request):
    data = [
        {
            'id': logbook.id,
            'likes': list(logbook.likes.values_list('username', flat=True))
        }
        for logbook in self.queryset if logbook.likes.exists()
    ]
    return Response(data)
```

원인 분석 2. Django ORM 처리 방식

1. 동기식 처리 + Django ORM은 DB 동기식 요청은 DB 연결을 블로킹하며 처리한다.
2. worker를 늘리면 DB 연결 수도 늘어나지만, DB가 동시에 처리할 수 있는 쿼리 수에는 한계가 있다.
3. DB가 포화 상태이면 추가 worker가 단순히 대기열을 늘리는 효과만 발생하고, 전체 처리량은 오히려 떨어질 수 있다.

원인 분석 3. 비동기 처리에서의 DB I/O 오버헤드

1. 동기식 처리에서는 요청이 들어오면 해당 worker 프로세스가 순차적으로 처리한다. (추가적인 스케줄링 로직이 없다)
2. 비동기식 처리에서는 요청이 들어오면 Future/Task 생성, 이벤트 루프 등록, Context 저장/복원 순서의 작업을 수행해야 한다.
3. N+1 쿼리 문제가 존재하여 DB 요청이 자주, 짧게 반복된다면 Async 스케줄링 비용이 더 커질 수 있다.
4. Django는 DB 접근이 기본적으로 동기식이며, 비동기식으로 처리한다면 worker 프로세스 내부에 DB 작업을 위한 동기식 스레드가 추가로 생성되고 관리되어야 한다.
5. Async 뷰와 Sync DB 스레드 간의 데이터 전달 및 통신 비용이 발생한다.

교훈 및 해결 방법

Async가 Sync보다 느린 것은 Async 환경이 DB 접근이라는 Sync 작업을 처리하면서 발생하는 이중 부화와 Async 스케줄링에서 오버헤드가 발생하기 때문이다.

DB I/O 병목을 완화하려면 **캐싱을** 적용하면 된다.

Redis 캐싱

Redis (Remote Dictionary Server)

- 인메모리 기반의 고성능 NoSQL 데이터베이스
- 키-값 구조의 데이터를 저장하고 관리
- 캐싱: 애플리케이션의 데이터베이스 부하를 줄이고 응답 속도를 높이기 위해 자주 사용됨

03

Redis 캐싱

/api/logbooks//likes/

```python
@action(detail=False, methods=['get'])  7개의 사용 위치(7개의 동적)  junnie082
def likes(self, request):
    cache_key = "logbook_likes" # 캐시 키 정의
    data = cache.get(cache_key)  # 캐시에서 먼저 조회
    if not data:
        data = [
            {
                'id': logbook.id,
                'likes': list(logbook.likes.values_list('username', flat=True))
            }
            for logbook in Logbook.objects.all().prefetch_related("likes")
            if logbook.likes.exists()
        ]
        cache.set(cache_key, data, timeout=60)  # Redis 캐시에 저장 (1분)
    return Response(data)
```

```python
class LikesAsyncView(APIView):  2개의 사용 위치  junnie082
    permission_classes = [IsAuthenticated]

    async def get(self, request):  junnie082
        cache_key = "logbook_likes" # 캐시 키 정의
        data = await sync_to_async(cache.get)(cache_key)  # async-safe 캐시 조회

        if not data:
            # DB 조회를 async-safe로 감싸기
            async_get_likes = sync_to_async(
                lambda: [
                    {
                        'id': logbook.id,
                        'likes': list(logbook.likes.values_list("username", flat=True))
                    }
                    for logbook in Logbook.objects.all().prefetch_related("likes")
                    if logbook.likes.exists()
                ],
                thread_sensitive=True
            )
            data = await async_get_likes()
            await sync_to_async(cache.set)(cache_key, data, 60)  # <- async-safe 캐시 저장

        return Response(data)
```

03

Redis 캐싱

```python
from celery import shared_task
from django.core.cache import cache
from .models import Logbook

@shared_task  2개의 사용 위치  junnie082
def add_like_task(logbook_id, user_id):
    log = Logbook.objects.get(id=logbook_id)
    log.likes.add(user_id)  # DB에 좋아요 추가
    cache.delete("logbook_likes")  # 캐시 무효화


@shared_task  2개의 사용 위치  junnie082
def remove_like_task(logbook_id, user_id):
    log = Logbook.objects.get(id=logbook_id)
    log.likes.remove(user_id)  # DB에서 좋아요 제거
    cache.delete("logbook_likes")  # 캐시 무효화
```

03

Redis 캐싱

worker = 4, vus = 50

```
ubuntu@ip-172-26-9-49:~/Backend/k6$ k6 run likes_sync.js

         /\      Grafana   /‾‾/
    /\  /  \     |\  __   /  /
   /  \/    \    | |/ /  /   ‾‾\
  /          \   |   (  |  (‾)  |
 / __________ \  |_|\_\  \_____/

     execution: local
        script: likes_sync.js
        output: -

     scenarios: (100.00%) 1 scenario, 50 max VUs, 1m0s max duration (incl. graceful stop):
              * default: 50 looping VUs for 30s (gracefulStop: 30s)


  TOTAL RESULTS

    checks_total.......: 1450    45.74304/s
    checks_succeeded...: 100.00% 1450 out of 1450
    checks_failed......: 0.00%   0 out of 1450

    ✓ login succeeded
    ✓ status was 200

    HTTP
    http_req_duration..............: avg=45.64ms min=17.87ms med=27.29ms max=644.36ms p(90)=74.61ms p(95)=123.82ms
      { expected_response:true }...: avg=45.64ms min=17.87ms med=27.29ms max=644.36ms p(90)=74.61ms p(95)=123.82ms
    http_req_failed................: 0.00%  0 out of 1450
    http_reqs......................: 1450   45.74304/s

    EXECUTION
    iteration_duration.............: avg=1.05s   min=1.01s   med=1.02s   max=1.72s    p(90)=1.07s   p(95)=1.12s
    iterations.....................: 1449   45.711493/s
    vus............................: 37     min=37        max=50
    vus_max........................: 50     min=50        max=50

    NETWORK
    data_received..................: 916 kB 29 kB/s
    data_sent......................: 156 kB 4.9 kB/s
```

```
ubuntu@ip-172-26-9-49:~/Backend/k6$ k6 run likes_async.js

         /\      Grafana   /‾‾/
    /\  /  \     |\  __   /  /
   /  \/    \    | |/ /  /   ‾‾\
  /          \   |   (  |  (‾)  |
 / __________ \  |_|\_\  \_____/

     execution: local
        script: likes_async.js
        output: -

     scenarios: (100.00%) 1 scenario, 50 max VUs, 1m0s max duration (incl. graceful stop):
              * default: 50 looping VUs for 30s (gracefulStop: 30s)


  TOTAL RESULTS

    checks_total.......: 1416    44.910379/s
    checks_succeeded...: 100.00% 1416 out of 1416
    checks_failed......: 0.00%   0 out of 1416

    ✓ login succeeded
    ✓ status was 200

    HTTP
    http_req_duration..............: avg=74.07ms min=29.62ms med=52.14ms max=3.46s p(90)=103.52ms p(95)=176.37ms
      { expected_response:true }...: avg=74.07ms min=29.62ms med=52.14ms max=3.46s p(90)=103.52ms p(95)=176.37ms
    http_req_failed................: 0.00%  0 out of 1416
    http_reqs......................: 1416   44.910379/s

    EXECUTION
    iteration_duration.............: avg=1.07s   min=1.03s   med=1.05s   max=4.46s p(90)=1.1s     p(95)=1.17s
    iterations.....................: 1415   44.878663/s
    vus............................: 26     min=26        max=50
    vus_max........................: 50     min=50        max=50

    NETWORK
    data_received..................: 1.4 MB 46 kB/s
    data_sent......................: 160 kB 5.1 kB/s
```

sync **http_req_duration... avg=45.64ms**

async **http_req_duration... avg=74.07ms**

03

Redis 캐싱 결과

worker = 4, vus = 50 일 때

- sync Redis 캐싱 적용 전: 152.9ms
- sync Redis 캐싱 적용 후: 45.64ms

- async Redis 캐싱 적용 전: 237.52ms
- async Redis 캐싱 적용 후: 74.07

03

Redis 캐싱 결론

서비스의 **병목은 CPU가 아닌 DB I/O 에서 오는 것이었으므로** 비동기/worker 수 늘려도 해결이 되지 않았다.
캐싱이 들어가는 순간 병목이 사라짐을 확인하였다.

또, **Django에서 Redis 캐싱 후 단순 조회 API는 동기(sync)가 비동기(async)보다 빠르다.**
async는 Django ORM 때문에 thread switching 오버헤드가 필요하기 때문이다.

“Fishial 외부 API 멀티프로세스 병렬처리”

04 Fishial 외부 API 멀티프로세스 병렬처리

```python
class FishialClient:
    AUTH_URL = "https://api-users.fishial.ai/v1/auth/token"
    UPLOAD_URL = "https://api.fishial.ai/v1/recognition/upload"
    RECOG_URL = "https://api.fishial.ai/v1/recognition/image"

    def __init__(self):
        self.client_id = settings.FISHIAL_CLIENT_ID
        self.client_secret = settings.FISHIAL_CLIENT_SECRET

    def get_token(self) -> str:
        r = requests.post(self.AUTH_URL, json={
            "client_id": self.client_id,
            "client_secret": self.client_secret
        })
        r.raise_for_status()
        return r.json()["access_token"]

    def request_signed_upload(self, token, filename, content_type, byte_size, checksum):
        headers = {
            "Authorization": f"Bearer {token}",
            "Content-Type": "application/json",
            "Accept": "application/json",
        }
        payload = {
            "blob": {
                "filename": filename,
                "content_type": content_type,
                "byte_size": byte_size,
                "checksum": checksum
            }
        }
        r = requests.post(self.UPLOAD_URL, headers=headers, json=payload)
        r.raise_for_status()
        return r.json()

    def recognize(self, token, signed_id):
        headers = {
            "Authorization": f"Bearer {token}",
            "Accept": "application/json",
        }
        params = {"q": signed_id}
        r = requests.get(self.RECOG_URL, headers=headers, params=params)
        r.raise_for_status()
        return r.json()
```

가장 처리가 복잡한 API

Fishial 에서 Access Token 발급 → 이미지 메타데이터 계산 (S3 이미지 다운로드) → Signed Upload 요청 → Direct Upload to S3 → Fishial Recognition (AI 연산 대기)

04 Fishial 외부 API 멀티프로세스 병렬처리

```python
class MockFishialClient:
    def __init__(self, shared_dict=None, lock=None, network_delay=0):
        self.shared_dict = shared_dict
        self.lock = lock
        self.network_delay = network_delay

    async def get_async_token(self):
        """[ASYNC] Step 1: 인증 토큰 발급 (공유 자원)"""
        # (비동기 환경에서는 멀티프로세싱 Lock/Manager를 직접 사용하지 않습니다.
        # 따라서 동기 실험과 달리 매번 토큰을 발급한다고 가정합니다.)
        await asyncio.sleep(0.1)
        return f"token_{uuid.uuid4().hex[:8]}"

    def get_sync_token(self):
        """[SYNC] Step 1: 인증 토큰 발급 (공유 자원)"""
        if self.lock and self.shared_dict is not None:
            if 'access_token' in self.shared_dict:
                return self.shared_dict['access_token']
            with self.lock:
                if 'access_token' in self.shared_dict:
                    return self.shared_dict['access_token']
                time.sleep(0.1)
                token = f"token_{uuid.uuid4().hex[:8]}"
                self.shared_dict['access_token'] = token
                return token
        else:
            time.sleep(0.1)
            return f"token_{uuid.uuid4().hex[:8]}"

    async def process_async_image_flow(self, token, image_index):
        """[ASYNC] Step 2~5: Presigned URL ~ 인공지능 분류 (핵심 병렬화 구간)"""
        if self.network_delay > 0:
            await asyncio.sleep(self.network_delay)

        # Step 2: Presigned URL 생성 요청
        await asyncio.sleep(0.05)
        # Step 3: 클라이언트 -> S3 업로드
        await asyncio.sleep(0.2)
        # Step 4: Fishial API 인식 요청
        await asyncio.sleep(0.5)

        return f"Image_{image_index}: Tuna"
```

Mock Fishial Client

- Fishial API 무료 호출 횟수 제한
- 실제 네트워크 통신에서 발생하는 느린 응답 시간을 시뮬레이션하여 (time.sleep() 및 asyncio.sleep() 활용) 테스트 함

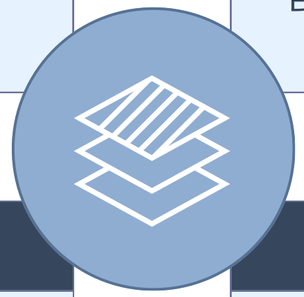
04 MockFishialClient 시뮬레이션 실험

순차 처리 (Sync)

작업을 하나씩 순서대로 처리

멀티 프로세싱 (Local Parallel)

4개의 CPU 코어를 사용하여 4개의 작업을 동시에 병렬 처리



MPI (분산 서버)

16개의 가상 노드/프로세스를 사용하여 최대한 많은 작업 동시에 처리

단일 스레드 논블로킹 (Async)

CPU 를 놀게 하지 않고 다른 작업을 대시 처리

순차 처리 (sync)

```
# ------------------------------------------------
# 1. 순차 처리 (Sequential)
# ------------------------------------------------
print(f"1️⃣  [순차 처리 - Sync] 실행 중...", end=" ", flush=True)
start = time.time()
client = MockFishialClient()
for i in range(NUM_IMAGES):
    token = client.get_sync_token()
    client.process_sync_image_flow(token, i)

t1 = time.time() - start
results['1. Sequential (Sync)'] = t1
print(f"완료! ({t1:.2f}s)")
```

```
def get_sync_token(self):
    """[SYNC] Step 1: 인증 토큰 발급 (공유 자원)"""
    if self.lock and self.shared_dict is not None:
        if 'access_token' in self.shared_dict:
            return self.shared_dict['access_token']
        with self.lock:
            if 'access_token' in self.shared_dict:
                return self.shared_dict['access_token']
            time.sleep(0.1)
            token = f"token_{uuid.uuid4().hex[:8]}"
            self.shared_dict['access_token'] = token
            return token
    else:
        time.sleep(0.1)
        return f"token_{uuid.uuid4().hex[:8]}"
```

```
def process_sync_image_flow(self, token, image_index):
    """[SYNC] Step 2~5: Presigned URL ~ 인공지능 분류"""
    if self.network_delay > 0:
        time.sleep(self.network_delay)

    # Step 2: Presigned URL 생성 요청
    time.sleep(0.05)
    # Step 3: 클라이언트 -> S3 업로드
    time.sleep(0.2)
    # Step 4: Fishial API 인식 요청
    time.sleep(0.5)

    return f"Image_{image_index}: Tuna"
```

멀티 프로세싱 (Local Parallel)

```
# ------------------------------------------------
# 2. 멀티프로세싱 (Local Parallel)
# ------------------------------------------------
print(f"2️⃣  [멀티프로세싱 (Lock)] {LOCAL_CORES}코어 실행 중...", end=" ", flush=True)
m = Manager()
shared_dict = m.dict()
lock = m.Lock()

start = time.time()
with Pool(processes=LOCAL_CORES) as pool:
    pool.starmap(worker_task_sync, [(i, shared_dict, lock, 0) for i in range(NUM_IMAGES)])

t2 = time.time() - start
results['2. Multi-Processing'] = t2
print(f"완료! ({t2:.2f}s)")
```

```
def worker_task_sync(image_idx, shared_dict, lock, network_delay=0):
    client = MockFishialClient(shared_dict, lock, network_delay)
    token = client.get_sync_token()
    client.process_sync_image_flow(token, image_idx)
    return True
```

MPI (분산 서버)

```
# ------------------------------------------------
# 3. 분산 서버 시뮬레이션 (MPI/Cluster)
# ------------------------------------------------
print(f"3️⃣  [분산 클러스터 (MPI)] {DISTRIBUTED_NODES}노드 실행 중...", end=" ", flush=True)

NETWORK_OVERHEAD = 0.05
dist_m = Manager()
dist_shared = dist_m.dict()
dist_lock = dist_m.Lock()

start = time.time()
with Pool(processes=DISTRIBUTED_NODES) as pool:
    pool.starmap(worker_task_sync,
                 [(i, dist_shared, dist_lock, NETWORK_OVERHEAD) for i in range(NUM_IMAGES)])

t3 = time.time() - start
results['3. Distributed Sys'] = t3
print(f"완료! ({t3:.2f}s)")
```

단일 스레드 논블로킹 (Async)

```
# ------------------------------------------------
# 4. Async 처리 (단일 스레드 논블로킹)
# ------------------------------------------------
print(f"4️⃣  [비동기 처리 (Async)] 단일 스레드 실행 중...", end=" ", flush=True)

start = time.time()

async def async_main():
    tasks = [worker_task_async(i) for i in range(NUM_IMAGES)]
    await asyncio.gather(*tasks)

# 파이썬 3.7+ 환경에서 asyncio.run()을 사용하여 실행
asyncio.run(async_main())

t4 = time.time() - start
results['4. Async (Single Thread)'] = t4
print(f"완료! ({t4:.2f}s)")
```

04 MockFishialClient 시뮬레이션 실험 결과

```
🧪 [실험 시작] 이미지 40장 처리

1️⃣  [순차 처리 - Sync] 실행 중... 완료! (34.59s)
2️⃣  [멀티프로세싱 (Lock)] 4코어 실행 중... 완료! (9.31s)
3️⃣  [분산 클러스터 (MPI)] 16노드 실행 중... 완료! (2.76s)
4️⃣  [비동기 처리 (Async)] 단일 스레드 실행 중... 완료! (0.86s)

============================================================
📊  [성능 비교 결과 그래프]
============================================================
1. Sequential (Sync)      | ████████████████████████████████ 34.59s
2. Multi-Processing       | ████████ 9.31s
3. Distributed Sys        | ███ 2.76s
4. Async (Single Thread)  |  0.86s
============================================================
```

04 MockFishialClient 시뮬레이션 실험 결과

방식	사용자원	소요 시간	성능 향상 배율 (vs 순차)
순차 처리 (Sync)	단일 스레드	34.59s	1.0 배
멀티프로세싱	4개 코어	9.31s	3.7 배
MPI	16개 노드	2.76s	12.5배
단일 스레드 논블로킹 (Async)	단일 스레드 (I/O 논블로킹)	0.86s	40.2배

04 Fishial 외부 API 멀티프로세스 병렬처리 결론

- 순차 처리: 모든 I/O 대기 시간이 누적되어 자원 낭비가 가장 심함
- 멀티 프로세싱: I/O 문제 미해결, CPU를 늘려도 여전히 대기 시간이 길어 비효율적
- MPI: 자원 과투입, 절대 속도는 빠르나 비싼 CPU를 I/O 대기에 낭비하여 비용 효율성 낮음
- **비동기 처리: 가장 성능이 좋음**, I/O 대기 시간에 다른 작업을 처리하여 자원 활용률이 40배 이상 높음

“

결론

”

05

결론 1

API 서버와 정적 파일을 분리하여 API 서버는 API 요청만 처리하도록 하는 것이 그렇지 않은 구조에 비해서 성능이 훨씬 좋다.

05

결론 2

DB 요청/처리를 하는 경우에는 비동기처리를 하거나 worker 를 늘리면 오히려 성능이 떨어지는 역효과가 나타난다. DB I/O 병목 현상이 발생하는 경우에는 캐싱을 적용함으로써 성능을 향상시킬 수 있다.

05

결론 3

복잡한 CPU 연산보다는 네트워크 I/O 요청이 많은 프로젝트에서는 자원을 많이 쓰는 병렬 처리보다 비동기 처리를 하는 것이 효과적이다.

05

프로젝트에서 배운 내용

상황에 맞는 기술 선택의 중요성

감사합니다

THANK YOU!