RabbitMQ + Kafka + Redis 실전 프로젝트

@미국달팽이

분산 시스템의 기초: Docker로 시작하는 실시간 아키텍처

RabbitMQ + Kafka + Redis + Flask 통합 환경 구축

5분 만에 완성하는 프로덕션급 개발 환경

분산 시스템의 기초: Docker로 시작하는 실시간 아키텍처

Docker Compose 파일 분석

```yaml
services:
  zookeeper:
    image: bitnami/zookeeper:latest
    ports:
      - "2181:2181"
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
    networks:
      - webnet

  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"    # AMQP 포트
      - "15672:15672"  # 관리자 인터페이스 포트
    networks:
      - webnet

  kafka:
    image: bitnami/kafka:latest
    ports:
      - "9092:9092"
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://localhost:9092
    networks:
      - webnet

  redis:
    image: redis/redis-stack-server:latest
    ports:
      - "6379:6379"    # Redis 기본 포트
      - "8001:8001"    # RedisInsight 웹 인터페이스 포트
    networks:
      - webnet

  postgres:
    image: postgres:latest
    environment:
      - POSTGRES_USER=admin
      - POSTGRES_PASSWORD=admin123
      - POSTGRES_DB=orders_db
    ports:
      - "5432:5432"
    networks:
      - webnet
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.5.0
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
    ports:
      - "9200:9200"
    networks:
      - webnet

networks:
  webnet:
    driver: bridge
```

분산 시스템의 기초: Docker로 시작하는 실시간 아키텍처

Flask & Celery 설정 단계

Flask 앱 초기화

```
from flask import Flask
from celery import Celery

app = Flask(__name__)
app.config['CELERY_BROKER_URL'] = 'amqp://rabbitmq'
celery = Celery(app.name, broker=app.config['CELERY_BROKER_URL'])
```

Celery Worker 실행:

- RabbitMQ 작업 처리

```
celery -A app.celery worker --loglevel=info -Q rabbit_orders
```

- Kafka 이벤트 전송

```
celery -A app.celery worker --loglevel=info -Q kafka_events
```

분산 시스템의 기초: Docker로 시작하는 실시간 아키텍처

실습 체크리스트

- RabbitMQ 관리자 화면 접속 (http://localhost:15672)
- Kafka Topic 생성 확인 (`docker exec -it kafka kafka-topics —list`)
- RedisInsight에서 Pub/Sub 채널 확인 (http://localhost:8001)

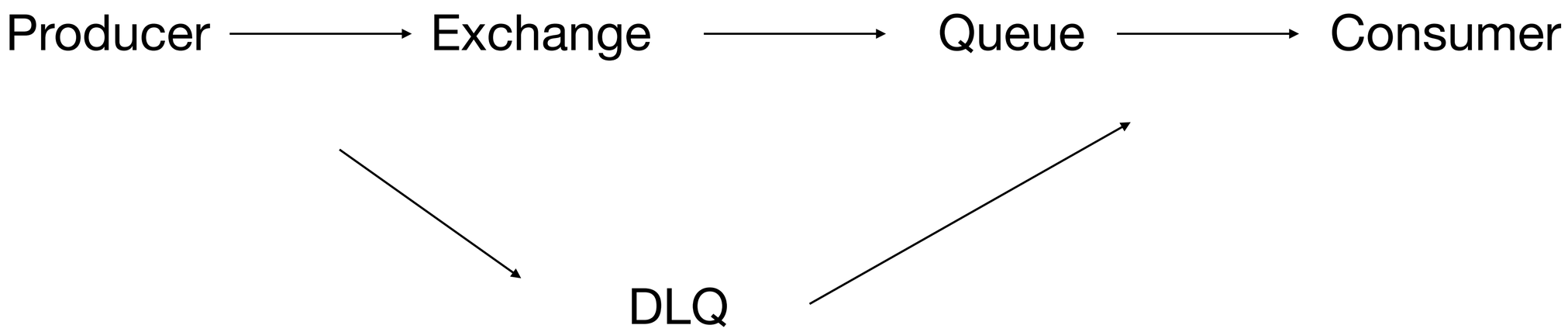
분산 시스템의 기초: Docker로 시작하는 실시간 아키텍처

추천 자료

- 공식 문서 링크: RabbitMQ TTL 설정, Kafka Exactly-Once 가이드, Redis Cluster 샤딩
- 도서: 《Designing Data-Intensive Applications》 (마틴 클레프만)

2. RabbitMQ - 비동기 작업 처리의 핵심 / RabbitMQ: 메시지의 운명을 설계하는 엔지니어링

Exchange 전략, DLQ, 재시도 정책으로 완성하는 프로덕션급 시스템



2. RabbitMQ - 비동기 작업 처리의 핵심 / RabbitMQ: 메시지의 운명을 설계하는 엔지니어링

Exchange 전략, DLQ, 재시도 정책으로 완성하는 프로덕션급 시스템

```
# 주문 ID로 라우팅 키 설정
channel.basic_publish(
    exchange='order_direct',
    routing_key='order_12345',
    body=message
)
```

Direct

주문 ID 기반 라우팅

실제 사례: 주문 상태 업데이트 (주문 ID별 정확한 라우팅)

```
# 모든 구독자에게 브로드캐스트
rabbitmqadmin publish exchange=inventory_fanout routing_key="" payload="ITEM_SOLD"
```

Fanout

알림 브로드캐스트

실제 사례: 재고 감소 이벤트를 결제/알림 시스템에 동시 전파


```
headers = {"priority": "high", "region": "asia"}
channel.basic_publish(
    exchange='priority_header',
    properties=pika.BasicProperties(headers=headers),
    body=message
)
```

Headers

메타데이터 기반 복잡 라우팅

실제 사례: 메타데이터 기반 라우팅 (예: `priority=high`)

```
# 패턴 기반 라우팅
channel.queue_bind(exchange='log_topic', queue='us_errors', routing_key='log.us.*')
```

Topic

카테고리별 분류 (e.g., `payment.#`)

실제 사례: 지역별 로그 분류 (`log.us.error`, `log.eu.info`)

3회 재시도 후 DLQ 전송 실습

2. RabbitMQ - 비동기 작업 처리의 핵심 / RabbitMQ: 메시지의 생명주기를 설계하는 법

Dead Letter Queue (DLQ) 설계

시나리오: 결제 실패 → 3회 재시도 → DLQ 전송

```
# RabbitMQ Queue 선언 시 DLQ 설정
channel.queue_declare(
    queue='payment_retry',
    arguments={
        'x-dead-letter-exchange': 'dlx',
        'x-message-ttl': 60000,  # 1분 후 DLQ 전송
        'x-max-retries': 3       # 최대 재시도 횟수
    }
)
```

2. RabbitMQ - 비동기 작업 처리의 핵심 / RabbitMQ: 메시지의 생명주기를 설계하는 법

실습: 결제 재시도 로직 구현

Consumer 예외 처리:

```python
def process_payment(message):
    try:
        charge_credit_card(message)
        message.ack()
    except PaymentError:
        message.reject(requeue=True)  # 재시도
```

DLQ 바인딩:

```
rabbitmqadmin declare exchange name=dlx type=direct
rabbitmqadmin declare queue name=dead_letters
rabbitmqadmin declare binding source=dlx routing_key=payment_retry
```

2. RabbitMQ - 비동기 작업 처리의 핵심 / RabbitMQ: 메시지의 생명주기를 설계하는 법

실습: 결제 재시도 로직 구현

Consumer 예외 처리:

```python
def process_payment(message):
    try:
        charge_credit_card(message)
        message.ack()
    except PaymentError:
        message.reject(requeue=True)  # 재시도
```

DLQ 바인딩:

```
rabbitmqadmin declare exchange name=dlx type=direct
rabbitmqadmin declare queue name=dead_letters
rabbitmqadmin declare binding source=dlx routing_key=payment_retry
```

2. RabbitMQ - 비동기 작업 처리의 핵심 / RabbitMQ: 메시지의 생명주기를 설계하는 법

에러 시나리오 & 해결책:

- **문제 1**: 메시지가 DLQ로 가지 않음
  원인: `x-dead-letter-exchange` 미설정 또는 TTL/재시도 조건 불충족
  해결: `rabbitmqctl list_queues arguments`로 설정 확인

- **문제 2**: 무한 재시도 루프
  원인: `requeue=True` 설정 시 max_retries 미적용
  해결: Consumer 레벨에서 재시도 카운터 구현

2. RabbitMQ - 비동기 작업 처리의 핵심 / RabbitMQ: 메시지의 생명주기를 설계하는 법

실습 체크리스트

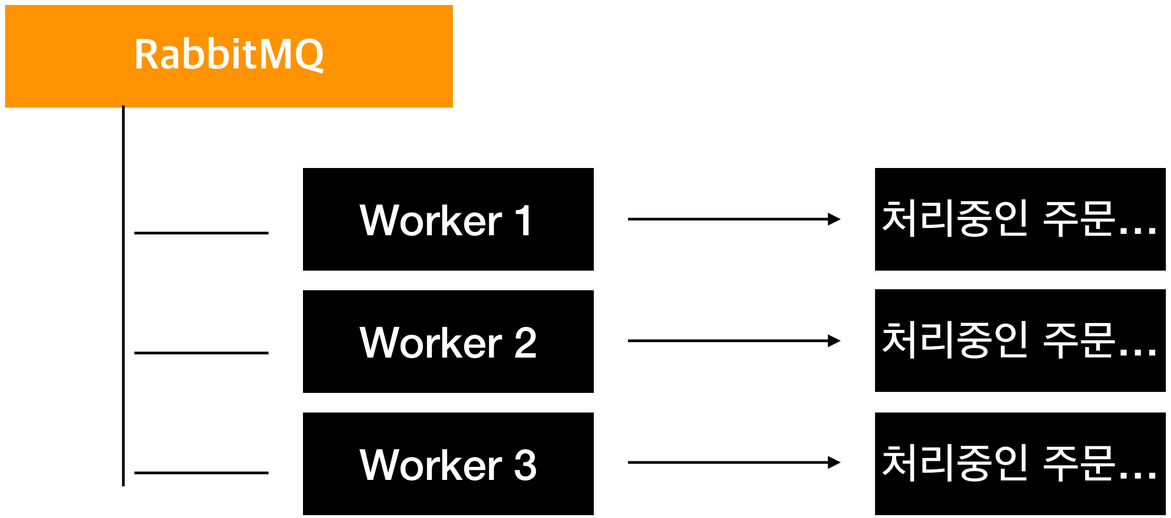
- Direct Exchange로 주문 메시지 라우팅 성공
- 결제 실패 시 3회 재시도 후 DLQ 도착 확인
- RabbitMQ 관리자 UI에서 DLQ 메시지 조회

3-1.고성능 주문 처리 파이프라인 구축

초고속 주문 처리의 비밀: Celery + Prefetch Count 마스터하기

"10,000개 주문을 10초 안에 처리하는 기술"

Celery Worker 병렬 처리 전략 **—concurrency=4 => 4개의 스레드**



3.고성능 주문 처리 파이프라인 구축

- **원칙 공식**
  최적 Prefetch Count = (Worker 수 × Concurrency) + 예비 버퍼

- **튜닝 전/후 비교**

  기본값: prefetch_count=1 → Worker가 1개 메시지만 가져와서 CPU 활용도가 낮음
  최적화: prefetch_count=10 → 각 Worker가 여러 메시지를 미리 가져와 CPU 사용률이 크게 향상됨

app.conf.worker_prefetch_multiplier = 10
app.conf.worker_max_tasks_per_child = 1000 # 메모리 누수 방지

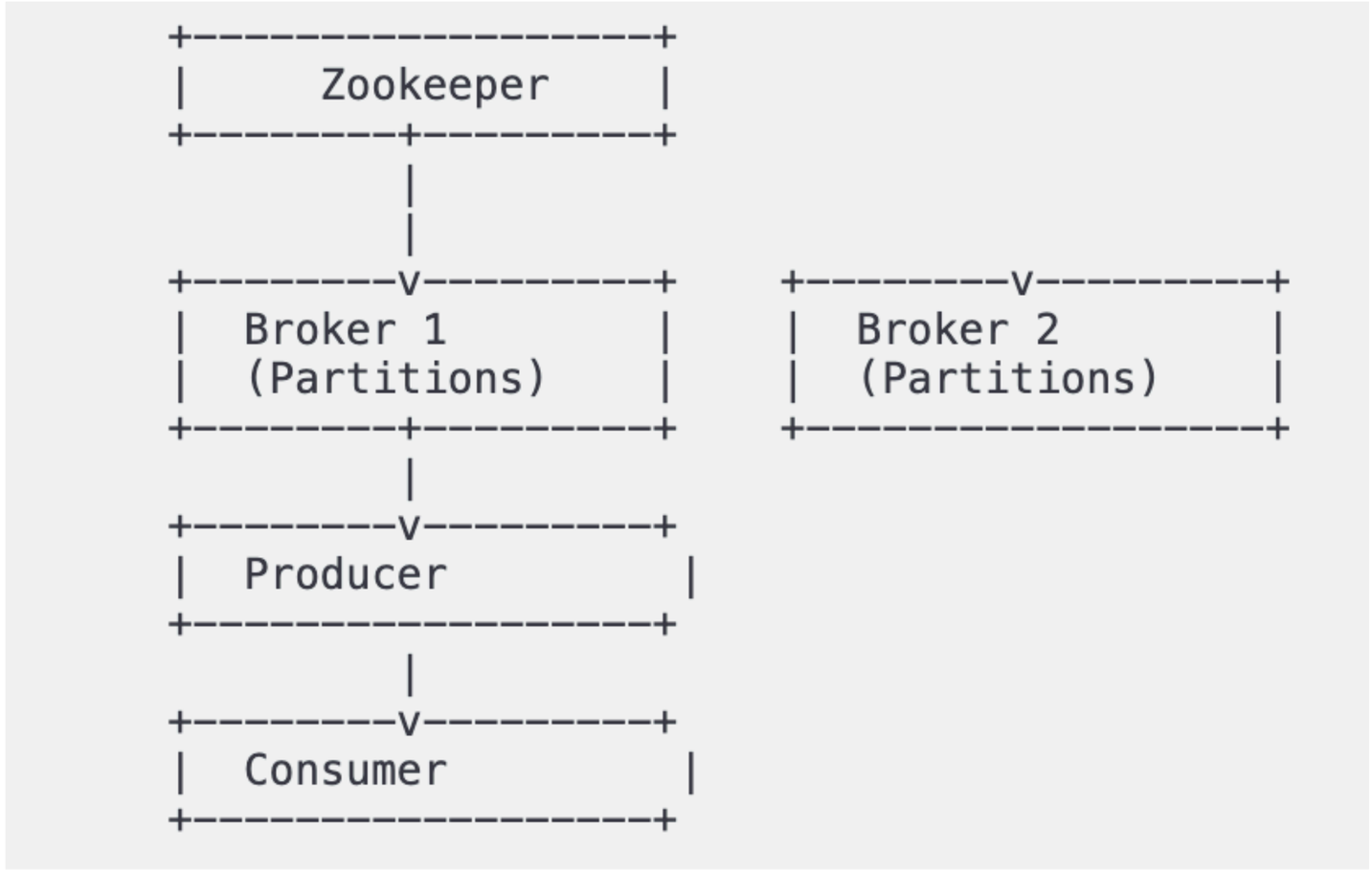
3.고성능 주문 처리 파이프라인 구축

체크리스트

- Celery Worker 4개가 정상적으로 실행되는지 확인
- Prefetch Count가 10으로 적용되었는지 점검
- 10,000건 주문 처리 시간이 15초 이내에 완료되었는지 확인

3.Kafka: 이벤트 드리븐 아키텍처의 심장

실시간 데이터 파이프라인 구축



3.Kafka: 이벤트 드리븐 아키텍처의 심장

실시간 데이터 파이프라인 구축

주요 주제:

1. Topic Partitioning 전략: Key-based 분산
2. Exactly-Once Semantic 보장 프로듀서 구현
3. 실습: 주문 이벤트 → 실시간 대시보드 연동 (Elasticsearch + Kibana)

3.Kafka: 이벤트 드리븐 아키텍처의 심장

왜 데이터를 쪼개야 하는가?

- 단일 디스크의 I/O 한계 → 분산 저장 필요성
- 같은 키는 항상 같은 파티션 (순서 보장)

파티션 = hash(메시지 키) % 파티션 수

```
# 주문 ID를 키로 해시 생성 후 특정 파티션에 메시지 전송
producer.send('orders', key=str(order_id), value=order_data)
```

3.Kafka: 이벤트 드리븐 아키텍처의 심장

Exactly-Once Semantic 보장 프로듀서 구현

중복 없이 정확하게 한 번만 처리하는 것이 왜 중요한가?

idempotence와 트랜잭션을 결합한 Exactly-Once 보증

```
producer = KafkaProducer(
    bootstrap_servers='kafka:9092',
    enable_idempotence=True,  # 중복 방지
    transaction_id='tx-1'     # 원자적 처리 보장
)

producer.begin_transaction()
producer.send('orders', value=order)
producer.commit_transaction()
```

3.Kafka: 이벤트 드리븐 아키텍처의 심장

트러블 슈팅 시나리오

만약 Consumer가 데이터를 따라가지 못해 lag가 발생한다면?

데이터 유실 문제가 있다면?

```
# 파티션 수 늘리기
kafka-topics --alter --partitions 10 --topic orders

# 데이터 유실 방지를 위한 프로듀서 설정
producer = KafkaProducer(acks='all')
```

3.스트리밍 데이터 처리 심화 Kafka Streams로 실시간 주문 집계

- Kafka Streams 개요 및 실시간 데이터 처리 개념
- Python Flask와 Kafka 연계 방안
- 장애 대응 전략
- 성능 튜닝 기법
- 실시간 매출 집계 및 1분 단위 TPS/이상치 탐지

3.스트리밍 데이터 처리 심화 Kafka Streams로 실시간 주문 집계

- **Kafka Streams란?**
  - Apache Kafka 기반의 스트림 처리 라이브러리
  - 실시간 데이터(무한 스트림) 처리에 적합

- **주요 개념**
  - 스트림(Stream)과 테이블(Table)
  - 배치 처리 vs. 스트림 처리

3.스트리밍 데이터 처리 심화 Kafka Streams로 실시간 주문 집계

- **연계 방법**
  - Python Kafka 클라이언트(kafka-python, confluent-kafka) 사용
  - Flask에서 Kafka 컨슈머를 통해 데이터 수신

- **통합 구조 예시**
  - Kafka Streams(Java) → 집계 결과를 Kafka 토픽에 기록 → Flask 웹 애플리케이션에서 구독 및 REST API 제공

- **추가 옵션**
  - 웹소켓 또는 SSE를 통한 실시간 데이터 푸시

3.스트리밍 데이터 처리 심화 Kafka Streams로 실시간 주문 집계

- **장애 감지**
  - 모니터링(컨슈머 lag, 스레드 상태, JMX)

- **장애 복구**
  - Kafka Streams의 자동 복구, 컨슈머 그룹 리밸런싱
  - 로컬 상태(State Store)와 체인지로그 활용

- **데이터 정합성 유지**
  - Exactly-Once 처리, 중복 감지 로직

3.스트리밍 데이터 처리 심화 Kafka Streams로 실시간 주문 집계

윈도우 크기 설정
- 텀블링 윈도우, 슬라이딩 윈도우
- 적절한 윈도우 크기 선택의 중요성

병렬 처리
- Kafka 파티션 수, 스트림 스레드 수 조정

Kafka 토픽 설정
- 복제, acks, 배치 크기, 메시지 압축

3.스트리밍 데이터 처리 심화 Kafka Streams로 실시간 집계 실습

TPS 계산
- 1분 텀블링 윈도우로 거래 건수 집계
- 평균 TPS = (분당 거래 건수 / 60)

이상치 탐지
- 임계치 기반: 사전 정의된 거래 수 기준 초과/미달 시 알림
- 통계적 방법: 평균, 표준편차 기반 이상 탐지 (예: 3시그마 규칙)

대응 방안
- 실시간 알람 및 모니터링 시스템 연계

4-1. Redis - 초고속 데이터 플랫폼, 캐시 전략으로 성능 극대화

Cache-Aside vs Write-Through 패턴 비교 및 실습

1.캐시 전략의 필요성과 기본 원리

2.Cache-Aside 패턴 vs Write-Through 패턴 비교

3.실습: 상품 조회 API 응답 시간 개선

4-1. Redis - 초고속 데이터 플랫폼, 캐시 전략으로 성능 극대화

왜 캐시가 필요할까?

- 데이터베이스 I/O 한계와 응답 시간 문제
- 메모리 기반 캐시(Redis)의 초고속 응답 특성
- 시스템의 근본 제약(디스크 I/O, 네트워크 지연)을 제거하는 방법

4-1. Redis - 초고속 데이터 플랫폼, 캐시 전략으로 성능 극대화

Cache-Aside 패턴

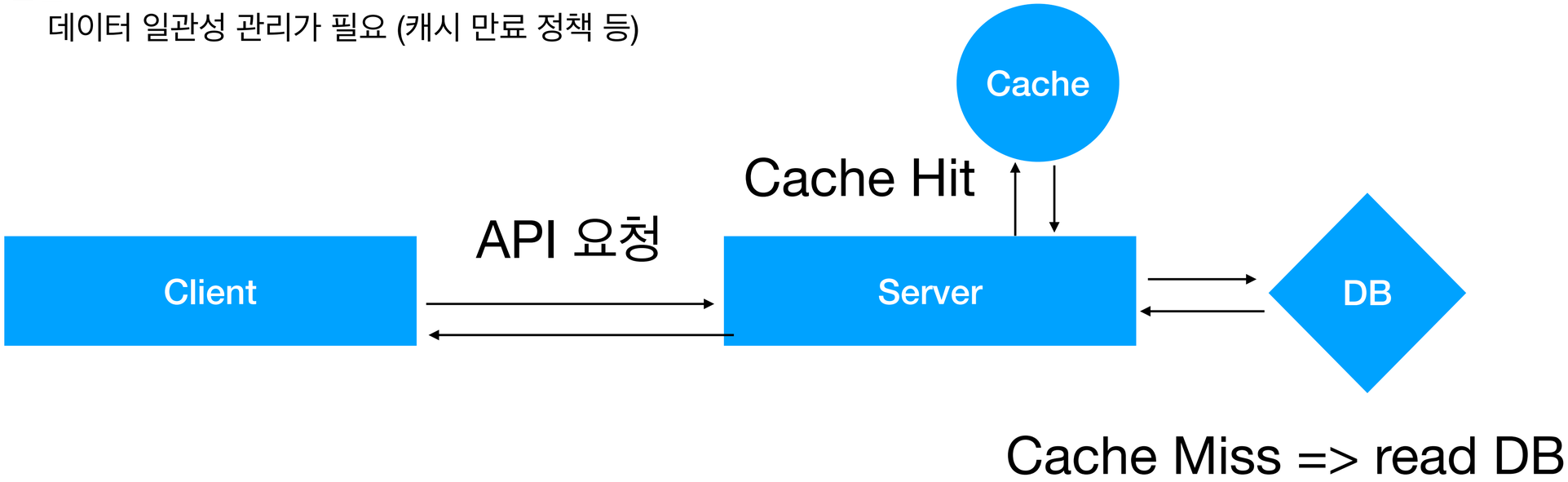
애플리케이션이 캐시를 직접 조회하고, 캐시에 없으면 DB에서 가져와 캐시에 저장하는 방식

장점:

구현이 간단하고, 캐시와 DB의 분리된 관리 가능
읽기 중심의 워크로드에 적합

단점:

데이터 일관성 관리가 필요 (캐시 만료 정책 등)



4-1. Redis - 초고속 데이터 플랫폼, 캐시 전략으로 성능 극대화

Write-Through 패턴

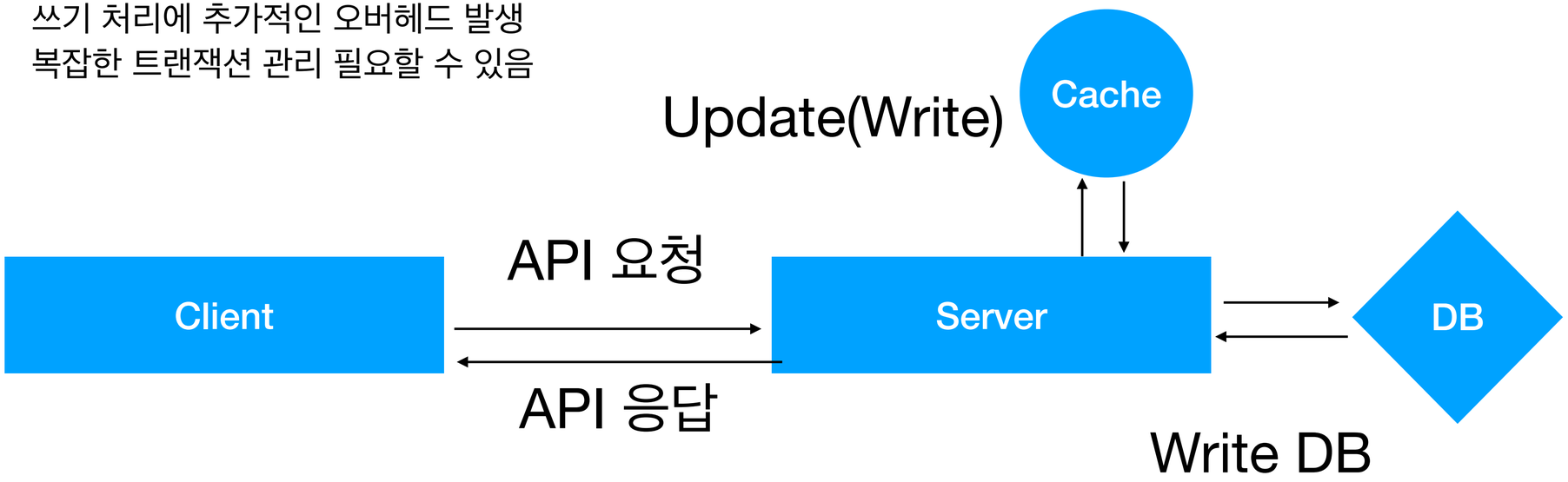
데이터가 쓰여질 때 동시에 캐시에 업데이트하는 방식

장점:

DB와 캐시의 데이터 일관성이 자동으로 유지
읽기 성능과 일관성 모두 보장

단점:

쓰기 처리에 추가적인 오버헤드 발생
복잡한 트랜잭션 관리 필요할 수 있음



4-1. Redis - 초고속 데이터 플랫폼, 캐시 전략으로 성능 극대화

실습: 상품 조회 API 응답 시간 개선

실습 목표

기존 0.5초의 응답 시간을 Redis 캐시 도입으로 10ms 이하로 개선

실습 단계

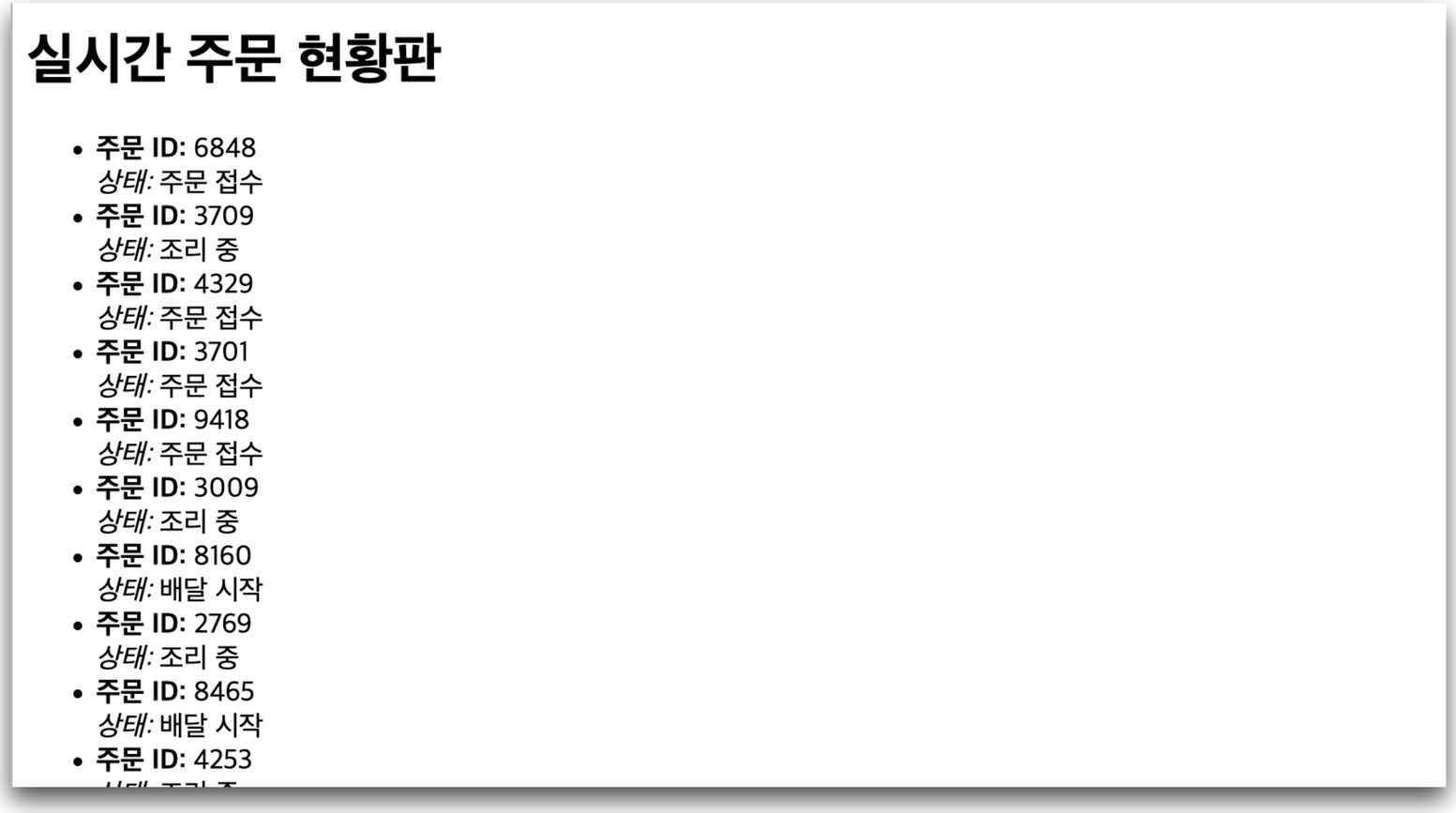
기존 상품 조회 API의 동작 원리 설명 (DB 직접 조회)
Cache-Aside 패턴 적용 예제 코드
    캐시 조회 → 캐시 미스 시 DB 조회 후 캐시에 저장
성능 개선 전후 응답 시간 비교 (벤치마크 결과 그래프)

체크리스트

캐시 조회/저장 로직 검증
캐시 만료 정책 설정 여부
응답 시간 벤치마크 결과 확인

4-2. Pub/Sub을 활용한 실시간 시스템

주문 상태 변경 이벤트 브로드캐스팅 및 실시간 현황판 구현

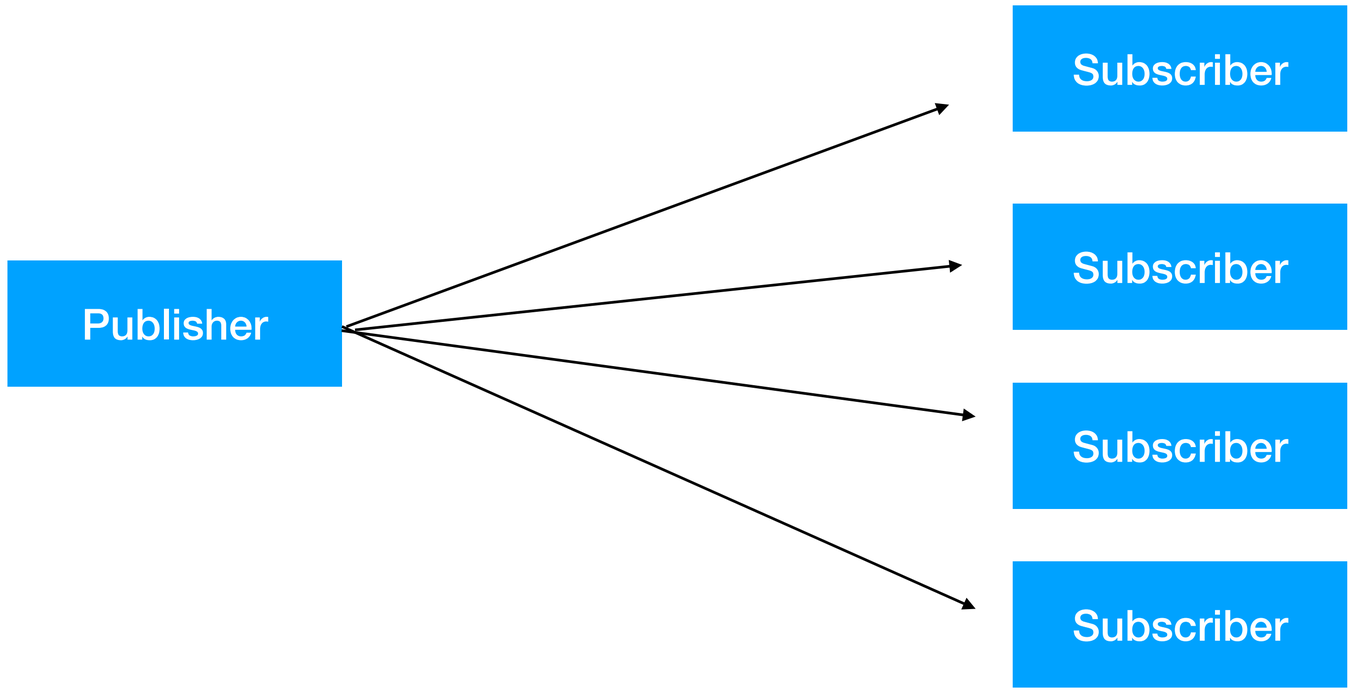


4-2. Pub/Sub을 활용한 실시간 시스템

왜 Pub/Sub 방식을 사용할까?

메시지를 한 번에 여러 곳으로 전달하는 구조의 장점
발행자와 구독자가 느슨하게 결합되어 있어 확장성과 유연성이 높음

데이터 전파의 근본 목적은 ‘정보의 즉각적 전달’이며, 이를 위해 중간에 불필요한 지연이 없어야 함



4-2. Pub/Sub을 활용한 실시간 시스템

실습: WebSocket + Redis Pub/Sub으로 실시간 주문 현황판 구현

실습 목표:

Redis Pub/Sub을 통해 주문 상태 변경 이벤트를 실시간으로 전파
WebSocket을 사용하여 클라이언트에 실시간으로 데이터를 푸시

실습 단계:

Redis에서 주문 이벤트 발행 코드 예제
WebSocket 서버와 Redis 구독자 연결 코드 예제
실시간 주문 현황판(HTML/JS)을 통해 이벤트 확인

체크리스트:

Redis Pub/Sub 채널에 메시지가 정상적으로 발행되는지
WebSocket 연결이 안정적으로 이루어지는지
클라이언트에서 실시간 데이터 수신 및 렌더링 여부

5-2. 장애 전파 방지 시스템 – Circuit Breaker 패턴과 Fallback 메커니즘 적용

왜 장애 전파 방지가 중요한가?

대규모 분산 시스템에서 한 구성요소의 실패가 전체 시스템에 미치는 영향

예시: 주문 처리 서비스와 결제 서비스 간 연쇄 실패

5-2. 장애 전파 방지 시스템 – Circuit Breaker 패턴과 Fallback 메커니즘 적용

Circuit Breaker 패턴 개념

Closed 상태: 모든 요청 통과, 실패 카운트 누적

Open 상태: 실패 임계치 초과 시 요청 차단

Half-Open 상태: 재시도 후 정상 복구 여부 확인

5-2. 장애 전파 방지 시스템 – Circuit Breaker 패턴과 Fallback 메커니즘 적용

PyBreaker를 통한 구현 예제

```
import pybreaker
import requests

# 3회 이상 연속 실패 시 60초간 회로 열림
circuit_breaker = pybreaker.CircuitBreaker(fail_max=3, reset_timeout=60)

@circuit_breaker
def call_external_service():
response = requests.get("https://api.example.com/data")
return response.json()
```

PyBreaker 라이브러리를 사용하여 외부 API 호출을 보호하는 코드 예제
실패 3회 시 회로 열림, 60초 후 재시도 → 성공 시 복구

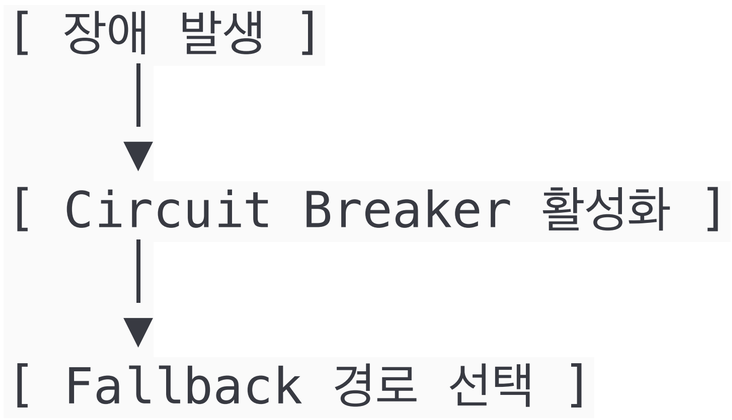
5-2. 장애 전파 방지 시스템 – Circuit Breaker 패턴과 Fallback 메커니즘 적용

장애 발생 시 대체 경로 확보

대체 서비스 호출: 캐시된 데이터, 이전 결제 결과 활용

메시지 큐 활용: RabbitMQ 또는 Kafka를 통한 비동기 재처리

캐싱 및 폴백: Redis를 통해 이전 데이터를 반환



5-2. 장애 전파 방지 시스템 – Circuit Breaker 패턴과 Fallback 메커니즘 적용

실제 구현 사례 및 연구 자료 분석

Netflix Hystrix 사례 소개

서비스 간 호출 시 Circuit Breaker를 적용하여 한 서비스의 장애가 전체 시스템에 영향을 미치지 않도록 설계

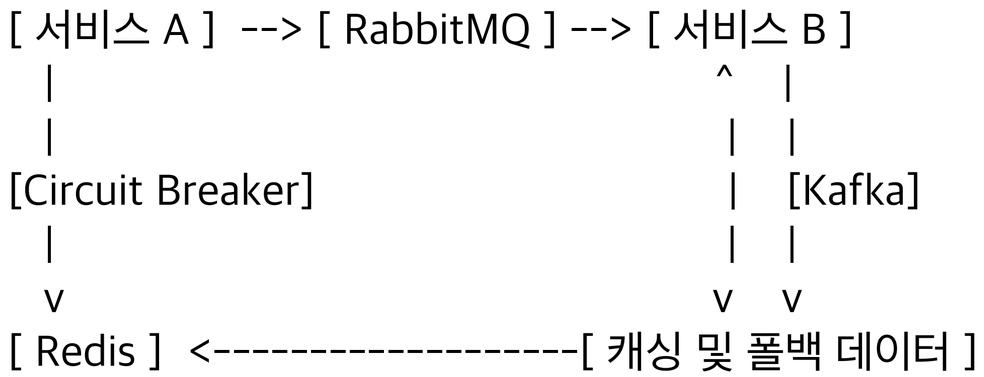
“전체 시스템의 탄력성*(resiliency)*을 높인다”

- 마틴 파울러

5-2. 장애 전파 방지 시스템 – Circuit Breaker 패턴과 Fallback 메커니즘 적용

종합 시스템 아키텍처

- RabbitMQ: 비동기 작업 큐로 장애 격리
- Kafka: 실시간 이벤트 스트리밍 및 모니터링
- Redis: 캐싱 및 빠른 폴백 데이터 제공
- Circuit Breaker: 서비스 호출 보호 및 장애 전파 차단



이 다이어그램은 서비스 A가 RabbitMQ를 통해 서비스 B로 비동기 작업을 전달하고, Kafka를 통해 실시간 이벤트를 스트리밍하며 Redis를 통해 캐싱 및 폴백 데이터를 관리하는 구조를 보여줌. Circuit Breaker는 서비스 간 호출에서 발생할 수 있는 장애를 차단.

5-2. 장애 전파 방지 시스템 – Circuit Breaker 패턴과 Fallback 메커니즘 적용

클라우드 및 쿠버네티스 적용 사례

- AWS 및 Kubernetes 배포 전략
- 관리형 서비스(예: Amazon MSK, ElastiCache) 활용
- 중앙화된 로그와 분산 트레이싱을 통한 장애 진단 및 복구