**서버 성능 최적화**

**서버 성능 기본 개념**

**성능 지표**

* ﻿﻿CPU 사용률 (CPU Utilization)
* ﻿﻿메모리 사용률 (Memory Usage)
* ﻿﻿디스크 i/O (Disk Input/Output)
* ﻿﻿네트워크 대역폭 (Network Bandwidth)
* ﻿﻿응답 시간 (Response Time)
* ﻿﻿트랜잭션 처리율 (Transactions Per Second, TPS)
* ﻿﻿에러율 (Error Rate)
* ﻿﻿업타임 (Uptime)
* ﻿﻿다운타임 (Downtime)

> 성능 지표에 따라 최적화해야 하는 아키텍쳐 구성 요소가 달라지게 됩니다.

**서버 성능 저하 원인**

* **﻿﻿하드웨어 자원 한계**
* ﻿﻿작업에서 필요로 하는 자원의 양이 서버의 CPU나 메모리 등의 자원 한계를 초과할 경우 성능 저하가 발생할 수 있습니다.
* ﻿﻿예를 들어, 과도한 트래픽이나 복잡한 계산 작업이 서버 자원을 과다하게 사용할 수 있습니다.
* ﻿﻿**네트워크 병목**
* ﻿﻿서버와 클라이언트 간의 통신 속도가 느려지거나, 네트워크 대역폭이 부족할 경우 요청 및 응답 전송에 지연이 발생하여 성능이 저하될 수 있습니다.
* ﻿﻿**코드 최적화 문제**
* ﻿﻿비효율적인 알고리즘 구현, 느린 데이터베이스 쿼리, 메모리 누수 등이 서버 성능 저하의 주요 원인이 될 수 있습니다.

﻿﻿이는 코드 리팩토링이나 데이터베이스 쿼리 튜닝을 통해 개선할 수 있습니다.

**서버 성능 최적화의 중요성**

* **﻿﻿사용자 만족도 향상**
* ﻿﻿빠르고 안정적인 서버 성능은 사용자 경험에 직접적인 영향을 미쳐, 사용자의 만족도를 높이는 데 중요합니다.
* ﻿﻿**비용 절감**
* ﻿﻿성능 최적화를 통해 하드웨어 및 인프라의 추가 비용을 줄일 수 있습니다.  효율적으로 자원을 활용함으로써 더 많은 요청을 처리할 수 있습니다.
* ﻿﻿**높은 가용성 및 신뢰성**
* ﻿﻿성능 최적화는 서버의 가용성과 신뢰성을 향상시키는 데 기여합니다.
* ﻿﻿부하 분산과 자원 관리를 효율적으로 수행함으로써 시스템이 더욱 안정적으로 운영될 수 있습니다.
* ﻿﻿**경쟁력 강화**
* ﻿﻿빠르고 효율적인 서버 성능은 기업의 경쟁력을 강화하는 데 중요한 요소입니다.

﻿﻿고객들에게 더 나은 서비스를 제공할 수 있어 시장에서 차별화된 위치를 유지할 수 있습니다.

애플리케이션 레벨 최적화

**1. 병목 지점 파악하기**

• 모니터링 도구를 통해 애플리케이션의 업타임과 자원 사용량을 모니터링합니다.

• 이 때, 특정 기능 및 요청에서 응답 시간이 얼마나 오래 걸리는지 확인할 수 있습니다.

* ﻿﻿혹은 자동화 테스트 도구 및 프로파일링 도구를 통해 특정 기능을 실행해 성능을 측정합니다.
* ﻿﻿**CPU 사용률 높음:** 특정 함수 또는 코드 블록에서 GPU가 지나치게 많이 사용되는 경우, 해당 부분을 최적화할 필요가 있습니다.
* ﻿﻿**메모리 누수:** 메모리 사용량이 지속적으로 증가하면 메모리 누수가 발생한 것일 수 있으며, 메모리 관리를 개선해야 합니다.

﻿﻿**i/0 병목:** 데이터베이스 쿼리가 너무 많거나 디스크 i/0가 높은 경우, 데이터 액세스를 최적화하거나 캐싱을 고려해야 할 수 있습니다.

**2. 코드 최적화**

비효율적인 코드를 개선하여 실행 시간을 단축하고 자원 사용을 최적화합니다.

* ﻿﻿알고리즘 변경: 보다 효율적인 알고리즘으로 교체하여 성능을 개선합니다.
* ﻿﻿**반복문 최적화:** 중복 계산을 줄이고 반복문을 최대한 단순화합니다.
* ﻿﻿**메모리 관리:** 메모리 누수를 방지하고 메모리 사용을 최적화합니다.
* ﻿﻿**비동기 처리:** i/0 작업이나 네트워크 호출과 같은 블로킹 작업을 비동기적으로 처리하여 CPU 자원을 효율적으로 활용할 수 있습니다.

﻿﻿**병렬 처리:** 여러 작업을 동시에 처리하여 전체 실행 시간을 단축하고, CPU 코어를 최대한 활용합니다.

**2. 코드 최적화: 알고리즘 성능 표기법**

* **﻿﻿Big-0 표기법**
* ﻿﻿Big O 표기법은 알고리즘의 시간 복잡도를 나타내는 데 사용되며, 입력 크기에 따라 알고리즘의 실행 시간이 어떻게 증가하는지를 설명합니다.
* ﻿﻿Big O는 최악의 경우를 기준으로 알고리즘의 성능을 분석합니다.
* ﻿﻿O(N) 과 같은 형식으로 표기합니다.
* ﻿**Omega) 표기법**
* ﻿﻿오메가 표기법은 최선의 경우 시간 복잡도를 나타내며, 알고리즘이 최소한 얼마나 빠를 수 있는지를 나타냅니다.
* ﻿오메가(N)과 같은 형식으로 표기합니다.
* ﻿﻿**(Theta) 표기법**
* ﻿세타 표기법은 평균 시간 복잡도를 나타내며, 알고리즘이 평균적으로 얼마나 오래 걸리는지를 나타냅니다.

세타(N)과 같은 형식으로 표기합니다.

**2. 코드 최적화: GC (Garbage Collector) 이해하기**

* ﻿﻿GC는 더 이상 사용되지 않는 객체를 자동으로 식별하고 제거하여 개발자가 명시적으로 메모리를 해제할 필요가 없게 해줍니다.
* ﻿﻿**GC의 주요 단계**
* ﻿﻿**Mark:** 사용 중인 객체와 사용되지 않는 객체를 식별
* ﻿﻿**Sweep:** 사용되지 않는 객체를 제거
* ﻿﻿**Compact:** 남은 객체들을 모아 메모리 단편화 방지 (선택적)
* ﻿﻿Serial, Parallel, CMS, G1,ZGC, Mark-and-Sweep 등 다양한 GC 알고리즘이 있습니다.

﻿﻿Java9+ 에서는 G1 알고리즘이 사용되고, Node js 에서는 Orinoco (Generational Garbage Collection) 알고리즘이 사용됩니다.

**2. 코드 최적화: 메모리 관리**

* ﻿﻿**객체 및 배열 사용 최적화**
* ﻿﻿**객체 및 배열 초기화:** 필요한 만큼만 객체와 배열을 초기화하고 사용합니다.  불필요한 미리 초기화나 크기를 예상하고 너무 큰 배열을 만드는 것은 메모리를 낭비할 수 있습니다.

﻿﻿**객체 및 배열 사용 후 메모리 해제:** 더 이상 사용하지 않는 객체나 배열은 null로 할당하여 메모리를 명시적으로 해제하거나, GC(Garbage Collector)에 의해 자동으로 해제될 수 있도록 합니다.

// 비효율적인 초기화

 let arr = new Array(1000000):

// 필요한 크기보다 큰 배열 초기화 // 최적화된 초기화

 let arr = :[]// 필요한 크기만큼 배열 초기화

**클로저 사용 최적화**

* ﻿﻿자바스크립트에서 클로저는 외부 함수의 변수를 계속 참조할 수 있어 메모리 누수가 발생할 수 있습니다. 필요한 경우에만 클로저를 사용하고, 불필요한 참조는 해제합니다.

﻿﻿자바에서는 내부 클래스를 사용할 때 외부 클래스의 참조를 가지게 되는 경우가 있습니다. 이 경우에도 필요 없는 경우에는 참조를 해제하여 메모리 누수를 방지할 수 있습니다.

**2. 코드 최적화: 비동기 처리**

* ﻿﻿비동기 처리는 작업을 순차적으로 처리하는 것이 아니라, 작업의 완료를 기다리지 않고 동시에 다음 작업을 수행하는 방식입니다.
* ﻿﻿이를 통해 네트워크 통신, 파일 입출력, 데이터베이스 액세스 등 I/O 작업을 효율적으로 처리하여 doqd 응답성을 향상시키고, 자원을 효율적으로 관리할 수 있습니다.

**비동기 처리의 필요성**

* **﻿﻿블로킹 방지**
* ﻿﻿일반적인 동기적 방식으로 작업을 처리하면 하나의 작업이 느린 I/0 작업으로 인해 모든 스레드가 대기 상태에 빠질 수 있습니다.
* ﻿﻿이를 비동기적으로 처리하면 블로킹을 방지하고 애플리케이션의 응답성을 유지할 수 있습니다.
* ﻿﻿**성능 개선**
* ﻿﻿비동기 처리를 사용하면 I/0 바운드 작업(네트워크 호출, 파일 입출력 등)이 발생할 때, 대기 시간 동안 다른 작업을 수행할 수 있습니다.

﻿﻿이는 전체적인 처리량과 성능을 개선하는 데 기여합니다.

**2. 코드 최적화: 비동기 처리 최적화 방법**

* **﻿﻿병렬 실행:** 독립적인 여러 비동기 작업을 동시에 실행합니다.
* ﻿﻿자바스크립트: Promise.all() 또는 Promise.allSettled0 사용
* ﻿﻿자바: Completableruture.allOf() 또는 병렬 스트림 사용
* ﻿﻿전체 실행 시간이 단축되고, 리소스 활용도를 향상시킬 수 있습니다.
* ﻿﻿단, 너무 많은 동시 실행은 시스템에 부하를 줄 수 있으므로 적절한 제한이 필요합니다.
* ﻿﻿**비동기 작업 체이닝 최적화:** 연속된 비동기 작업들을 효율적으로 연결하여 처리합니다.
* ﻿﻿Promise 체이닝 , CompletableFuture 의 thenCompose 메서드 활용하여 비동기 작업의 체이닝을 작성합니다.

﻿﻿체이닝 최적화를 통해 코드 가독성을 향상 시키고, 복잡한 비동기 흐름 관리를 용이하게 할 수 있습니다.

**서버 레벨 최적화**

**서버 설정 최적화 (Nginx, Apache 등)**

Nginx, Apache와 같은 웹 서버의 설정을 최적화하여 전체적인 성능을 개선합니다.

* ﻿﻿**버퍼 및 연결 설정:** 클라이언트 요청을 처리하는 버퍼 크기와 동시 연결 수를 조정하여 서버 응답 속도를 최적화합니다.
* ﻿﻿**Keep-Alive 설정**: Keep-Alive를 적절히 활용하여 여러 요청을 하나의 TCP 연결로 관리하여 네트워크 지연을 줄입니다.
* ﻿﻿**캐싱 설정:** 정적 파일 캐싱을 통해 반복적인 파일 요청에 대한 서버 부하를 감소시키고 응답 속도를 높입니다.

• **압축 설정:** Gzip 등의 압축 알고리즘을 사용하여 전송 데이터를 압축하여 대역폭을 절감하고 네트워크 성능을 개선합니다.

* ﻿﻿**요청 및 응답 버퍼 크기 조정:** 버퍼의 크기가 너무 작으면 디스크 I/0 가 증가하고, 너무 크면 메모리가 낭비됩니다.

﻿﻿**연결 타임아웃 조정:** 연결 유지 시간이 너무 짧으면 불필요한 재연결이 자주 발생하고, 너무 길면 유휴 연결로 인해 리소스가 낭비됩니다.

**로드 밸런싱**

* ﻿﻿로드 밸런싱은 여러 서버나 네트워크 장치에 가해지는 부하를 분산시켜 서비스의 가용성과 성능을 향상시키는 기술입니다.
* ﻿﻿**하드웨어 및 소프트웨어 로드 밸런싱을** 통해 여러 서버 간에 트래픽을 분산하여 서비스의 가용성과 성능을 향상시킵니다.
* ﻿﻿**스티키 세션**을 활용하여 동일한 사용자 요청이 항상 같은 서버로 전달되도록 유지하여 세션 관리의 복잡성을 줄일 수 있습니다.
* ﻿﻿**주요 목표**
* **﻿﻿부하 분산:** 네트워크 트래픽이나 작업 부하를 여러 대의 서버에 골고루 분배하여 단일 서버에 집중되는 부하를 줄입니다.
* ﻿﻿**고가용성:** 하나 이상의 서버가 다운되더라도 다른 서버가 요청을 처리하여 서비스 중단을 최소화합니다.

﻿﻿**응답 시간 개선:** 사용자 요청이 처리되는 시간을 단축하여 빠른 응답을 제공합니다.

**로드 밸런서 종류**

* **﻿﻿하드웨어 로드 밸런서**
* ﻿﻿전용 하드웨어 장비를 사용하여 로드 밸런싱을 수행합니다.
* ﻿﻿높은 처리량과 성능을 제공하지만 비용이 높고 확장성이 제한될 수 있습니다.
* ﻿﻿**소프트웨어 로드 밸런서**
* ﻿﻿소프트웨어로 구현된 로드 밸런싱 기능을 가진 서버 프로그램을 사용합니다.
* ﻿﻿가상 머신, 컨테이너, 클라우드 인스턴스 등에서 실행할 수 있어 확장성이 좋습니다.
* ﻿﻿Nginx, Apache 등을 통해 로드 밸런싱을 수행할 수 있습니다.
* ﻿**DNS 기반 로드 밸런서**
* ﻿﻿DNS 서버에서 클라이언트의 요청을 여러 서버 IP 주소로 매핑하여 부하를 분산합니다.

• 간단하고 비용이 저렴하지만, 서버의 상태를 실시간으로 감지하지 못하고 부하 분산을 정교하게 조정할 수 없는 단점이 있습니다.

**Nginx에서 로드 밸런싱 설정하기**

1. Upstream 블록 정의하기: Nginx에서는 upstream 블록을 사용하여 백엔드 서버 그룹을 정의합니다. 이 그룹에는 로드 밸런싱이 적용될 서버들의 주소와 포트가 포함됩니다.

2. 로드 밸런싱 옵션 추가하기: 정의한 upstream 블록에 로드 밸런싱 알고리즘과 기타 옵션을 추가할 수 있습니다. 기본적으로 Nginx는 Round Robin 방식을 사용합니다.

**설정 상세 설명**

* ﻿﻿upstream 블록: servers upstream 블록 내에서 localhost를 사용하여 로컬 서버에 대한 서버 그룹을 정의합니다.

﻿﻿proxy\_pass 설정: location / 블록 내에서 proxy\_pass 지시어를 사용하여 실제 요청을 처리할 서버 그룹을 지정합니다. 위 예시에서는 servers 를 지정하여 이 로컬 서버 그룹의 서버들에게 요청을 전달합니다.

**서버 컴퓨팅 리소스 관리**

* **﻿﻿CPU 관리 및 최적화:** 서버의 CPU 사용률을 모니터링하고, 다중 코어를 효율적으로 활용하여 작업 처리 능력을 최대화합니다.
* ﻿﻿**멀티스레딩 및 멀티프로세스:** 다중 스레드 및 프로세스를 이용하여 병렬 처리를 촉진하고 CPU 자원을 최대한 활용합니다.
* ﻿﻿**최적화된 코드 실행:** JIT 컴파일러와 같은 기술을 사용하여 CPU가 더 빠르게 코드를 실행할 수 있도록 최적화합니다.
* ﻿﻿**메모리 관리:** 메모리 사용량을 모니터링하고, 가비지 컬렉션을 조정하여 메모리 누수를 방지하고 서버의 안정성을 유지합니다.
* ﻿﻿**힙 및 스택 최적화:** 메모리 할당 및 해제를 최적화하여 메모리 관리 오버헤드를 줄이고 성능을 향상시킵니다.
* ﻿﻿**디스크 I/0 관리:** 디스크 접근 속도를 향상시키기 위해 SSD 등의 고성능 스토리지를 사용하거나 RAID 설정을 최적화합니다.

﻿﻿**캐싱 전략:** 서버와 데이터베이스 사이의 I/0 비용을 줄이기 위해 적절한 데이터 캐싱 전략을 도입하여 데이터 접근 속도를 높입니다.

**데이터베이스 최적화**

* **﻿﻿인덱스 최적화**
* ﻿﻿적절한 인덱스를 사용하여 데이터베이스 질의의 속도를 높입니다.
* ﻿﻿자주 사용되는 필드에 인덱스를 추가하여 검색 및 정렬 작업을 최적화합니다.
* ﻿﻿**쿼리 최적화**
* ﻿﻿쿼리 튜닝: 비효율적인 쿼리를 개선하고, 조인 및 서브쿼리를 최적화하여 데이터베이스 부하를 출입니다.
* ﻿﻿**버퍼 풀 및 캐시 관리**
* ﻿﻿버퍼 풀 크기 조정: 데이터베이스 캐시 메모리를 적절히 할당하여 자주 액세스되는 데이터를 메모리에 유지합니다.
* ﻿﻿쿼리 캐싱: 동일한 쿼리의 결과를 캐시하여 데이터베이스 부하를 줄이고 응답 시간을 단축시킵니다.
* ﻿﻿**분할 및 파티셔닝**
* ﻿﻿대용량 테이블을 파티션화하여 데이터를 물리적으로 분리하고 질의 속도를 향상시킵니다.

﻿﻿시간 기반 파티셔닝을 사용하여 데이터를 시간별로 분할하여 데이터 유지 관리를 용이하게 합니다.

**클라우드 환경 최적화**

**스케일 아웃 (Scale Out) 과 스케일 업 (Scale Up)**

* ﻿﻿**스케일 아웃 (Scale Out)**
* ﻿﻿스케일 아웃은 기존 시스템에 새로운 인스턴스를 추가하거나, 기존 인스턴스를 복제하여 트래픽을 분산시키는 것을 말합니다.
* ﻿﻿트래픽이 증가하거나 서비스의 부하를 분산시키기 위해 사용됩니다.
* ﻿﻿**장점**
* ﻿﻿**확장성:** 트래픽 증가에 따라 필요한 만큼의 자원을 추가할 수 있습니다.
* ﻿﻿**고가용성:** 여러 인스턴스를 통해 장애 발생 시 서비스 중단을 최소화할 수 있습니다.

> **스케일 아웃은 수평적 확장으로, 서버 인스턴스를 여러 대로 늘려서 트래픽을 분산시키는 방식입니다.**

**오토 스케일링 (Auto Scaling)**

* ﻿﻿오토 스케일링은 클라우드 환경에서 자동으로 서버 인스턴스의 수를 조정하여 트래픽 변동에 효율적으로 대응하는 기술입니다.
* ﻿﻿주로 웹 애플리케이션과 같은 서비스에서 사용되며, 사용량에 따라 자동으로 인스턴스를 추가하거나 제거하여 서비스의 가용성을 유지하고 비용을 절감하는 목적을 가지고 있습니다. (Scale Out)

**동작 원리**

* ﻿﻿**트리거 설정:** 오토 스케일링은 설정된 트리거를 기반으로 동작합니다. 주로 CPU 사용률, 메모리 사용량, 네트워크 트래픽 등의 지표가 사용됩니다.
* ﻿﻿**예시:** CPU 사용률이 70%를 넘으면 자동으로 인스턴스를 추가하고, CPU 사용률이 일정 기준 이하로 내려가면 인스턴스를 줄입니다.
* ﻿﻿**스케일링 액션:** 트리거가 발생하면 오토 스케일링 서비스는 다음과 같은 액션을 수행합니다.
* ﻿﻿**자동 스케일 아웃**: 추가 인스턴스를 시작하여 현재 트래픽 수요에 맞춥니다.
* ﻿﻿**자동 스케일 인:** 트래픽이 감소하면 더 이상 필요하지 않은 인스턴스를 자동으로 종료하여 비용을 절감합니다.

**• 스케일링 그룹:** 인스턴스를 관리하는 단위로, 하나 이상의 서버 인스턴스를 포함합니다. 스케일링 그룹은 동일한 애플리케이션 또는 서비스에 속한 인스턴스들을 관리하며, 고가용성을 보장하고 로드 밸런싱을 수행합니다.

**오토 스케일링 (Auto Scaling)**

* **﻿﻿장점**
* ﻿﻿**자원 최적화:** 트래픽에 따라 유동적으로 인스턴스 수를 조정하여 자원을 최적화합니다.
* ﻿﻿**가용성 향상:** 자동으로 인스턴스를 추가하여 서비스의 가용성을 높입니다.
* ﻿﻿**비용 절감:** 필요에 따라 인스턴스를 추가하거나 제거하여 클라우드 리소스를 효율적으로 사용합니다.
* ﻿﻿**주의사항**
* **﻿﻿트리거 설정:** 정확한 트리거 설정이 필요합니다. 너무 높은 트리거 설정은 초과 비용을 발생시킬 수 있으며, 너무 낮은 설정은 성능 저하를 초래할 수 있습니다.
* ﻿﻿**테스트:** 오토 스케일링 설정 전에 충분한 테스트와 모니터링 계획이 필요합니다. 예기치 않은 스케일링 액션을 방지하기 위해 경험과 데이터를 기반으로 설정해야 합니다.

**> 적절한 테스트와 모니터링을 통해 클라우드 서비스의 비용을 최적화해야 합니다**.

**컨테이너화 및 오케스트레이션**

* **﻿﻿컨테이너는** 애플리케이션과 그 의존성을 함께 묶어 격리된 환경에서 실행할 수 있게 하는 가상화 기술입니다.
* ﻿﻿주요 구성 요소
* ﻿﻿**이미지(Image)**
* ﻿﻿컨테이너 실행에 필요한 파일 시스템과 애플리케이션 코드를 포함한 불변 (Immutable) 패키지입니다.
* ﻿﻿이미지에는 애플리케이션의 실행 환경과 의존성이 포함되어 있습니다.
* ﻿﻿**컨테이너(Container)**
* ﻿﻿이미지를 실행한 상태로, 애플리케이션이 실행되는 격리된 환경입니다.
* ﻿﻿컨테이너는 일회성으로 실행되며, 종료되면 상태가 유지되지 않습니다.
* ﻿﻿**레지스트리(Registry)**
* ﻿﻿이미지를 저장하고 배포하는 저장소입니다.
* ﻿﻿**Dockerfile**
* ﻿﻿이미지를 생성하기 위한 설정 파일입니다.

﻿﻿명령어를 통해 이미지 빌드 과정을 정의합니다.

**컨테이너화 및 오케스트레이션**

* ﻿﻿컨테이너의 주요 특징
* ﻿﻿**격리성:** 각 컨테이너는 별도의 프로세스와 파일 시스템을 사용하여 독립된 환경을 제공합니다. 이를 통해 애플리케이션 간의 간섭을 최소화합니다.
* ﻿﻿**이식성:** 컨테이너는 운영 체제 수준에서 가상화되므로, 동일한 컨테이너 이미지를 어디서나 실행할 수 있습니다. 이를 통해 개발 환경과 운영 환경의 차이를 출입니다.
* ﻿﻿**경량성:** 컨테이너는 호스트 운영 체제의 커널을 공유하므로, 전통적인 가상 머신보다 훨씬 가벼우며 빠르게 시작할 수 있습니다.

﻿﻿**신속한 배포:** 컨테이너는 빠르게 생성, 삭제, 배포할 수 있으며, 애플리케이션의 지속적 배포(Continuous Deployment)와 통합(Continuous Integration)에 적합합니다.

**컨테이너화 및 오케스트레이션**

**컨테이너 오케스트레이션은** 여러 컨테이너의 배포, 관리, 확장, 네트워킹, 로깅 등을 자동화하는 프로세스를 의미합니다. 이는 수많은 컨테이너가 복잡하게 얽혀 있는 현대 애플리케이션 환경에서 필수적인 도구로, 주로 Kubernetes와 Docker Swarm과 같은 플랫폼이 사용됩니다.

* ﻿﻿주요 기능
* ﻿﻿**자동 배포 및 복구:** 컨테이너화된 애플리케이션을 자동으로 배포하고, 장애가 발생할 경우 자동으로 복구합니다.
* ﻿﻿**확장 및 축소:** 수요에 따라 컨테이너 인스턴스를 자동으로 확장(Scale Out) 하거나 축소(Scale In)합니다.
* ﻿﻿**리소스 할당:** CPU, 메모리, 스토리지 등 자원을 컨테이너에 효율적으로 할당하여 성능을 최적화합니다.

﻿﻿**로드 밸런싱:** 트래픽을 여러 컨테이너 인스턴스로 분산시켜 애플리케이션의 가용성과 성능을 향상시킵니다.

**Kubernetes**

* ﻿﻿쿠버네티스 (Kubernetes, ks) 는 가장 널리 사용되는 컨테이너 오케스트레이션 도구로, 60ogle이 개발하여 오픈 소스로 공개했습니다.
* ﻿﻿자동 배포, 확장, 복구, 로드 밸런싱, 비밀 관리, 네임스페이스로 자원 격리 등 다양한 기능을 제공합니다.
* ﻿﻿구성 요소
* ﻿﻿**마스터 노드:** 클러스터를 관리하는 역할을 하며, API 서버, 스케줄러, 컨트롤러 매니저 등이 포함됩니다.

﻿﻿**워커 노드**: 실제 애플리케이션 컨테이너가 실행되는 노드로, Kubelet, Kube-Proxy, 컨테이너 런타임 등이 포함됩니다.

**Kubernetes 최적화하기**

* **﻿﻿클러스터 설정 최적화:** 클러스터의 노드 수와 크기를 적절히 설정하여 워크로드를 효율적으로 분산합니다.
* ﻿﻿**오토 스케일링:** Horizontal Pod Autoscaler(HPA)를 사용하여 트래픽 증가 시 자동으로 파드를 확장하고, 트래픽 감소 시 축소합니다.
* ﻿﻿**리소스 요청 및 제한 설정:** 각 파드의 CPU와 메모리 요청(request) 및 제한(limit)을 설정하여 자원 분배를 최적화합니다.
* ﻿﻿**네임스페이스 활용:** 네임스페이스를 통해 리소스를 격리하고 관리하여 성능을 최적화합니다.

﻿﻿**네트워크 최적화:** 네트워크 플러그인(CNI)을 사용하여 네트워크 성능을 최적화하고, 서비스 간 트래픽을 최적화합니다.

**CDN (Content Delivery Network)**

**CDN**

CDN은 전 세계적으로 사용자에게 웹 콘텐츠를 빠르게 전송하는 서비스입니다. 주로 정적 파일(이미지, CSS, 자바스크립트 파일 등)을 효율적으로 제공하고, 대규모 트래픽 처리와 성능 최적화를 위해 사용됩니다.

주요 구성 요소

* ﻿﻿**Edge Servers:** CDN의 핵심 구성 요소로, 전 세계 여러 지역에 분포된 서버입니다. 이 서버들은 콘텐츠를 캐싱하고 사용자에게 제공합니다.
* ﻿﻿**Origin Server:** 원본 서버는 CDN에 의해 캐싱되는 콘텐츠의 출처가 되는 서버입니다. CDN이 원본 서버에서 콘텐츠를 가져와 캐싱합니다.

﻿﻿**DNS 서비스:** CDN은 DNS를 통해 사용자의 위치를 식별하고 가장 가까운 CDN 서버로 리디렉션합니다.

**CDN의 주요 기능과 원리**

**• 캐싱:** CDN은 전 세계에 분산된 서버 네트워크를 통해 콘텐츠를 캐싱합니다. 이렇게 함으로써 원본 서버에 직접 접근하는 횟수를 줄이고 사용자의 요청에 빠르게 응답할 수 있습니다.

**• 근접성 최적화:** CDN 서버는 사용자에게 더 가까운 위치에 배치되어 있어 물리적 거리에 따른 지연을 줄이고, 더 빠른 콘텐츠 전송을 가능하게 합니다.

**• 부하 분산:** CDN은 원본 서버의 부하를 줄이기 위해 트래픽을 여러 CDN

서버로 분산시킵니다. 이로 인해 원본 서버는 더 많은 동시 요청을 처리할 수 있습니다.

**• 보안:** CDN은 DDOS 공격으로부터 보호할 수 있는 기능을 제공할 수 있으며, SSL 인증서를 통한 암호화 서비스도 제공할 수 있습니다.