[1장] 사용자 수에 따른 규모 확장성

가상 면접 사례로 배우는 대규모 시스템 설계 기초

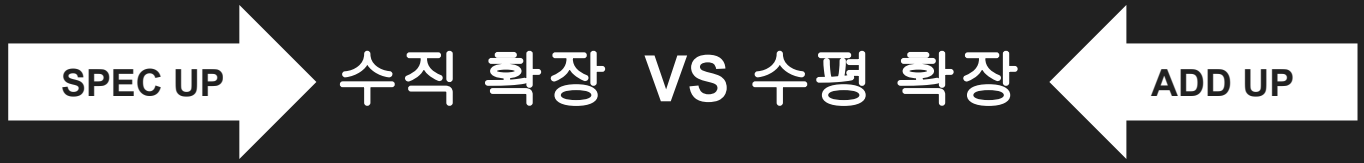
이인호 / mystyle2006

확장성(Scalability)이란?

- 시스템의 성장 능력을 설명하며, 사용자 요청이 증가할 때 필요한 컴퓨팅 자원과 서비스를 쉽게 늘리거나 줄일 수 있는 능력을 말한다.

얼마나 정확하고 빠르게 효율적으로 확장할 수 있으냐? 가 중요

확장의 방식



이러한 컴퓨팅 자원과 서비스를 높은 확장성을 가지기 위해 뭘 해야할까요?

첫번째, 모듈화된 아키텍처

소프트웨어를 모듈화를 잘하여 각 어플리케이션들이 독립적으로 개발, 테스트, 배포할 수 있도록 하는 것입니다. 이렇게 모듈화를 통해 필요한 부분만 수정할 수 있고 다른 부분에 영향을 미치지 않을 수 있어요.

두번째, 자동화된 테스트와 배포

테스트 자동화 및 지속적인 통합/지속적인 배포(CI/CD) 프로세스를 구축하여 변경사항을 안정적으로 테스트하고 배포할 수 있어요.

세번째, 클라우드 기술과 컨테이너 기술

AWS, GCP 등 다양한 클라우드 기술을 활용하여 확장성있는 설계를 할 수 있어요. 아마 이러한 확장성이 높기 때문에 스타트업 뿐만아니라 다양한 대기업에서도 클라우드 기술을 적극적으로 도입하려고 하는 것 같아요. 컨테이너 기술 또한 확장하는 하는데 아주 유용하죠.

네번째, 모니터링과 분석

1장 타이틀에서 알 수 있듯이 규모에 따라 인프라는 변화해야합니다. 따라서 확장성있는 설계를 위해서는 모니터링과 분석은 필수 요소인 것 같아요. (책에도 나온 것처럼!)

마지막으로, 연속적인 개선과 반복

무엇을 해야하기 전에 일단 해야 무엇을 해야할지 알 수 있죠. 소프트웨어 개발 및 운영 프로세스를 지속적으로 개선하고 반복함으로써 높은 수준의 확장성을 달성할 수 있을거에요.

확장성에 함께 따라오는 탄력성(Elasticity)

- 확장성을 찾아보다보니 탄력성(Elasticity)이라는 단어가 자주 보였는데요. 간단하게 비교하여 알아보겠습니다.

탄력성(Elasticity)이란?

- 시스템이나 소프트웨어가 변화나 부하의 증감에 유연하게 대응할 수 있는 능력을 말한다. 이는 시스템이 변화에 적응하거나 부하가 증가했을 때 추가 리소스를 할당하여 성능을 유지하는 데 사용된다.

확장성(Scalability)와 탄력성(Elasticity) 비슷한거 같은데... 뭐가 다를까?

- 확장성: 부하로 인한 시스템의 크기 조절에 초점에 초점!
- 탄력성: 부하로 인한 시스템이 동적으로 변화하는 환경에 대응하는 능력에 초점! (실시간)

좀 더 쉽게 이해기 위해 사례들을 보면

- 확장성이 높은 사례
  - 대규모 데이터베이스 시스템 (예: Google의 Bigtable, Amazon의 DynamoDB)
    - 대규모 데이터베이스 시스템은 막대한 양의 데이터를 처리하고 저장해야 합니다. 이러한 시스템은 데이터를 여러 노드에 분산하여 처리하므로, 데이터베이스의 크기가 증가하면 시스템을 확장하여 새로운 노드를 추가할 수 있습니다.
  - 대규모 웹 애플리케이션 (예: Facebook, Twitter)
    - 대규모 웹 애플리케이션은 많은 사용자 요청을 처리하고 매우 높은 트래픽을 관리해야 합니다. 확장성이 높아야 합니다. 이러한 웹 애플리케이션은 일반적으로 수평적 확장을 사용하여 여러 서버 인스턴스를 추가하여 트래픽을 분산하고 처리합니다.

좀 더 쉽게 이해기 위해 사례들을 보면

- 탄력성이 높은 사례
  - 클라우드 컴퓨팅 플랫폼 (예: AWS, Azure, Google Cloud)
    - 클라우드 컴퓨팅 플랫폼은 탄력성이 높은 전형적인 예입니다. (그래서 AWS에서 Scalable 이 아닌 Elastic이라고 브랜딩함) 사용자는 필요에 따라 서버 인스턴스, 스토리지, 데이터베이스, 그리고 다양한 클라우드 서비스를 신속하게 프로비저닝할 수 있습니다. 이러한 플랫폼에서는 자동 확장 기능을 통해 부하가 증가하면 자동으로 인프라를 확장하여 추가 요청을 처리할 수 있습니다.

  - 인터넷 스트리밍 서비스 (예: Netflix, YouTube):
    - 인터넷 스트리밍 서비스는 많은 동시 사용자와 다양한 컨텐츠에 대한 요청을 처리해야 합니다. 이러한 서비스는 탄력성이 높아야 합니다. 사용자 수에 따라 동적으로 확장하여 서버 및 네트워크 리소스를 할당합니다. 또한 캐싱 및 콘텐츠 전달 네트워크(CDN)와 같은 기술을 사용하여 사용자에게 최적의 성능을 제공합니다.

유사하지만 차이는 존재한다.
하지만 좋은 설계를 위해서는 둘을 떼놓을 수 없을 듯하다.

규모에 따른 확장 변화

단일 서버 -> 데이터베이스 분리 -> 로드 밸런서로 수평 확장 -> 데이터베이스 분산 처리 (Replication) -> 캐시 서버 분리 -> 정적 컨텐츠 캐시(CDN) 분리 -> Stateless 아키텍처 -> 지리적 분산 -> 각 애플리케이션 의존성 분리 (메시지 큐) -> 로깅 및 CI/CD 추가 -> 데이터베이스 수평 확장 (샤딩) -> 그 이상…

규모에 따라 변화하는 과정에서 공통점을 찾을 수 있다.
바로 수직적 확장보다는 수평적 확장을 선택한다는 것이다.

수직적 확장을 선택하지 않는 이유?

- **CPU, RAM** 등 성능을 무한히 증설할 수 없다.
- **SPOF(Single Point Of Failure,** 단일 장애점) 에 매우 노출되어 있다.
- 비용이 많이 든다. 고성능으로 갈수록 단가가 높아짐!

하지만! 실제 서비스를 운영하다보면 수평적 확장을 미리 준비하지 못했다면 (확장성이 낮다면) 수직적 확장이 클라우드 환경에서는 가장 빠르게 대응할 수 있는 옵션이기에 절대 잊어버리면 안된다.

수평적 확장의 장점은 책에서도 나와 알고 있지만 수평점 확장에도 단점이 존재할까?

- 복잡성 증가: 수평적 확장은 여러 서버로 분산시키는 것을 의미하기에 분산한 만큼 관리하는데 복잡성을 증가시킨다. 그렇기 때문에 수평적 확장을 선택했다면 로깅과 모니터링을 더 신경써야한다.
- 샤딩 오버헤드: 데이터의 샤딩(수평적 확장)은 책에서 간략하게 나온 것처럼 데이터의 적재를 얼마나 효율적으로 적재하느냐에 따라 성능이 달라진다. 이 때 데이터의 분배 과정에서 오버헤드가 발생할 수 있다.
- 일관성과 동기화 문제: 분산 시스템에서는 여러 서버 간의 데이터 일관성을 유지하는 것이 매우 어렵다. 서버 간의 동기화 문제와 일관성을 유지하기 위한 복잡한 매커니즘이 필요할 수 있다.
- 비용 문제

분산 처리만으로 모든 트래픽을 처리하기 비효율적이기 때문에 캐싱으로 보완

수직, 수평 확장만으로 모든 트래픽을 처리해야하는 것은 매우 비효율적이다. 우리도 일을 하다보면 반복적인 것을 함수로 만들거나 상수로 정의하듯이 캐시를 통해 자주 참조되는 자원들을 대신 처리해 줄 캐시 서버 또는 **CDN**을 활용한다.

이러한 캐시를 활용할 때 주의해야할 점은?

- 캐시는 어떤 상황에 바람직한가?
- 어떤 데이터를 캐시에 두어야 하는가?
- 캐시에 보관된 데이터는 어떻게 만료(expire) 되는가?
- 일관성(consistency)는 어떻게 유지되는가?
- 장애에는 어떻게 대처할 것인가?
- 캐시 메모리는 얼마나 크게 잡을 것인가?
- 데이터 방출(eviction) 정책은 무엇인가?

이러한 캐시를 활용할 때 주의해야할 점은?

- 캐시는 어떤 상황에 바람직한가?
- 어떤 데이터를 캐시에 두어야 하는가?
- 캐시에 보관된 데이터는 어떻게 만료(expire) 되는가?
- 일관성(consistency)는 어떻게 유지되는가?
- 장애에는 어떻게 대처할 것인가?
- 캐시 메모리는 얼마나 크게 잡을 것인가?
- 데이터 방출(eviction) 정책은 무엇인가?

정적 컨텐츠는 CDN으로 delivery 하기

- 주로 AWS 상에서 정적 컨텐츠를 빠르고 효율적으로 제공하기 위해 CloudFront를 활용한다.

이러한 CDN에는 주의해야할 점은 무엇일까?

- 비용
  - 자주 사용하지 않는 것은 구지 CDN에 적재하는 것은 비용만 발생!
- 적절한 만료 시한 설정
- CDN 장애 대처 방안
- 컨텐츠 무효화 (Invalidation)

비동기 아키텍처로 어플리케이션 간의 의존성을 줄여 확장성을 높일 수 있다.

복잡한 서비스의 경우 서로의 의존성이 점점 높아지는 경우가 많다. 이렇게 의존성이 높아지면 한 곳의 장애가 전체 장애로 퍼지는 **SPOF** 와 유사한 장애가 발생할 수도 있기 때문에 메시지 큐를 활용하여 좀 더 안정적인 서비스를 구성할 수 있다.

하지만 비동기 아키텍처의 경우 어떤 점을 주의해야할까?

비동기 아키텍처 주의해야할 점

복잡한 서비스의 경우 서로의 의존성이 점점 높아지는 경우가 많다. 이렇게 의존성이 높아지면 한 곳의 장애가 전체 장애로 퍼지는 **SPOF** 와 유사한 장애가 발생할 수도 있기 때문에 메시지 큐를 활용하여 좀 더 안정적인 서비스를 구성할 수 있다.

감사합니다