마이크로 서비스 아키텍쳐
(Micro Service Architecture)

2014년12월
채문창

모놀리틱 아키텍쳐 (Monolithic Architecture)

- 현재 까지의 전통적인 웹 개발 방식
- 하나의 애플리케이션에 모든 로직 포함
- 상호 호출 (Call By Reference)

모놀리틱 아키텍쳐 단점

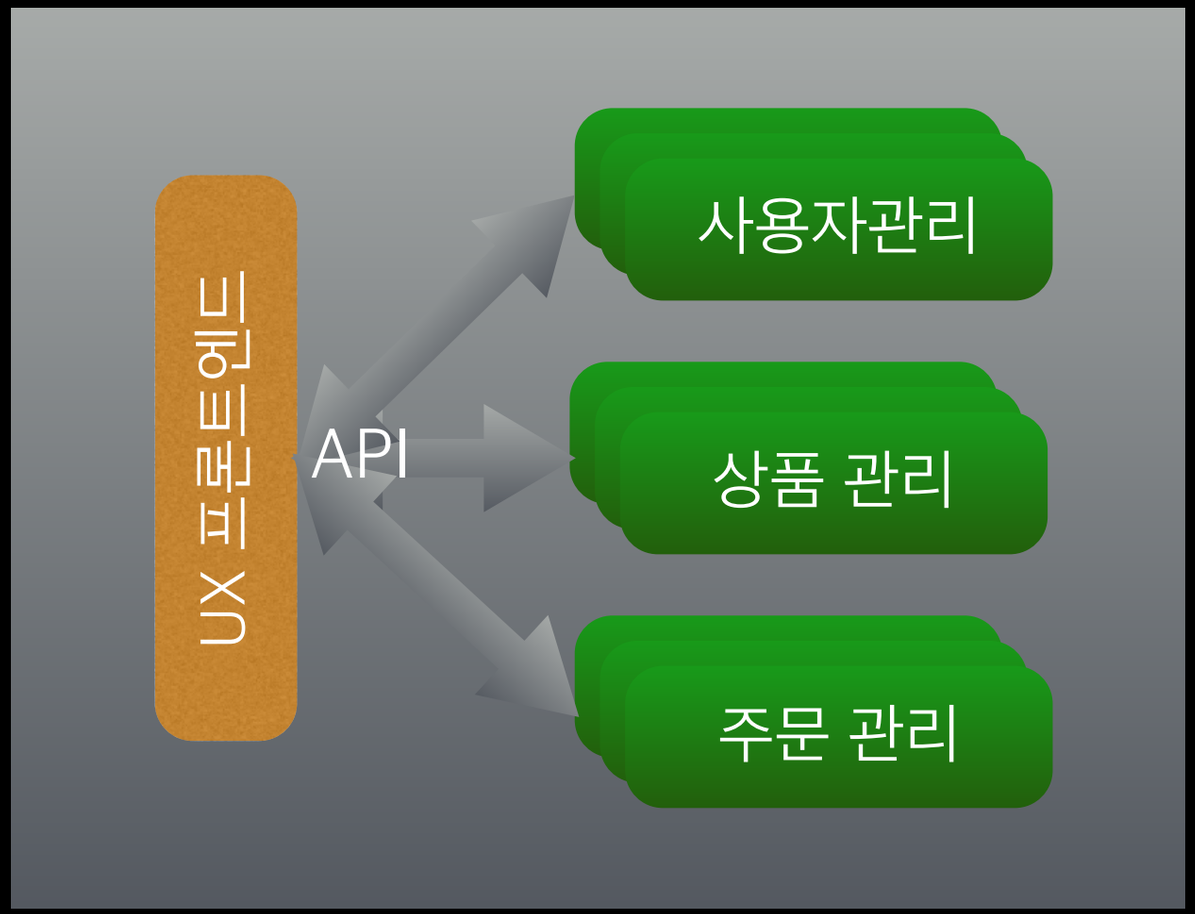
- 단일 시스템의 확장 문제
- 다양한 기능의 독립 개발 문제
- 배포 및 서비스 적용 문제
- 코드 수정에 따른 부작용이 전체 시스템에 영향
- Call-by-Reference 의 강력한 결합 문제

MSA 정의

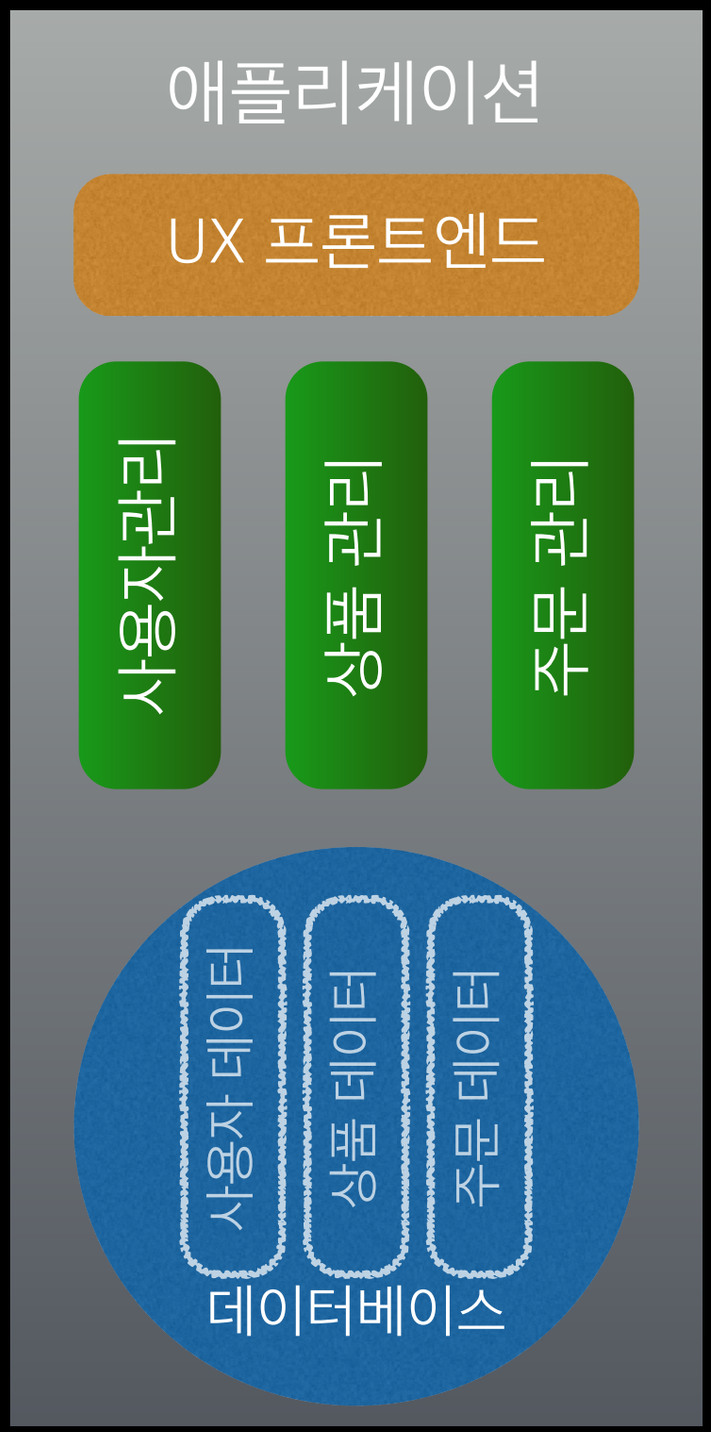
- 서비스 디자인 스타일
- 작은 서비스의 결합을 통해 하나의 응용프로그램 개발
- 각각의 서비스는 독립적인 비즈니스 로직
- 완전 자동화된 개발/배포환경에 의해 각각 독립적으로 배포
- 최소한의 중앙 관리 체계
- 멀티 프로그래밍 언어, 멀티 데이터스토리지 기술로 작성 가능
- 모놀리틱 아키텍쳐 반대

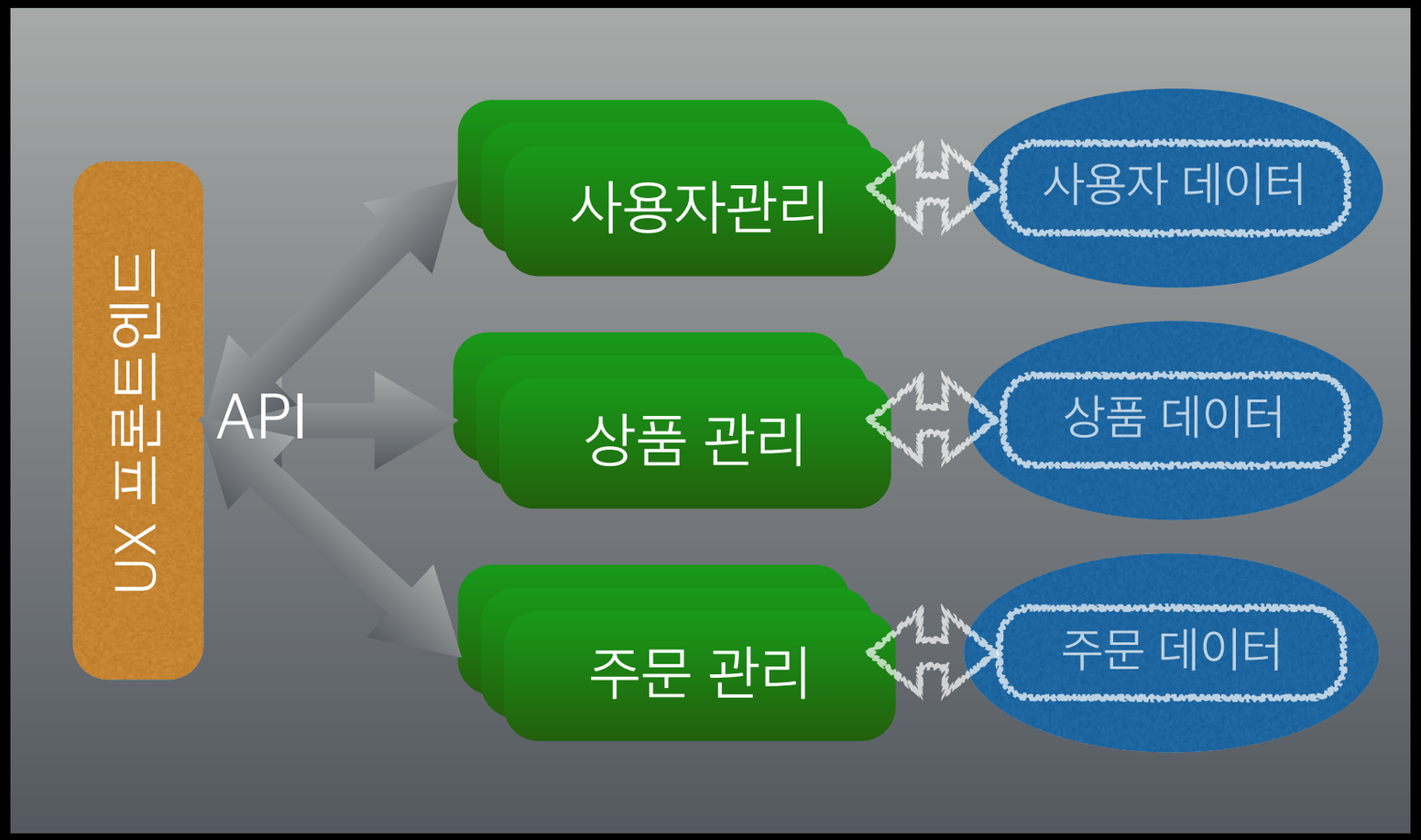
모놀리틱 아키텍쳐 차이


애플리케이션
UX 프론트엔드
사용자관리
상품관리
주문관리
데이터베이스


UX 프론트엔드
API
사용자관리
상품 관리
주문 관리

모놀리틱 저장소 차이


애플리케이션
UX 프론트엔드
사용자관리
상품관리
주문관리
사용자 데이터
상품 데이터
주문 데이터
데이터베이스


UX 프론트엔드
API
사용자관리
상품 관리
주문 관리
사용자 데이터
상품 데이터
주문 데이터

MSA 특징

- 단독으로 실행 가능
- 독립적으로 배포 가능
- 시스템 규모와 복잡성에 대한 관리
- 서비스를 충분히 작은 크기로 나누어 개발
- 상호 연계를 통해 좀 더 복잡하고 거대한 시스템 생성
- 서비스의 양과 복잡성 뿐만이 아니라 스케일링에도 높은 자유도
- 클라우드 컴퓨팅이나 고확장성 시스템의 요구조건에 정확히 부합

MSA 장점

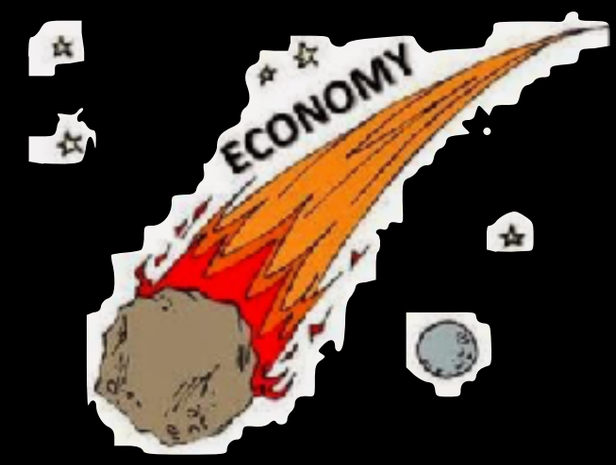
- 각각의 마이크로서비스는 심플하며, 각각의 비즈니스 요구사항에 특화
- 각각의 마이크로서비스는 개별 팀에서 독립적으로 개발/배포가 가능
- 각각의 마이크로서비스는 다른 프로그래밍언어, 다른 도구를 사용하여 개발 (polyglot)
- 각각의 마이크로서비스는 다른 데이터 저장소를 사용할 수 있으며 서로 느슨하게 연결
- 고속 개발에 최적화 (DevOps와 결합되어 개발속도와 개선에 있어서 높은 효용성)
- 자동화된 유닛 테스트와 시나리오테스트로 빠른 배포주기에도 불구 뛰어난 품질 유지

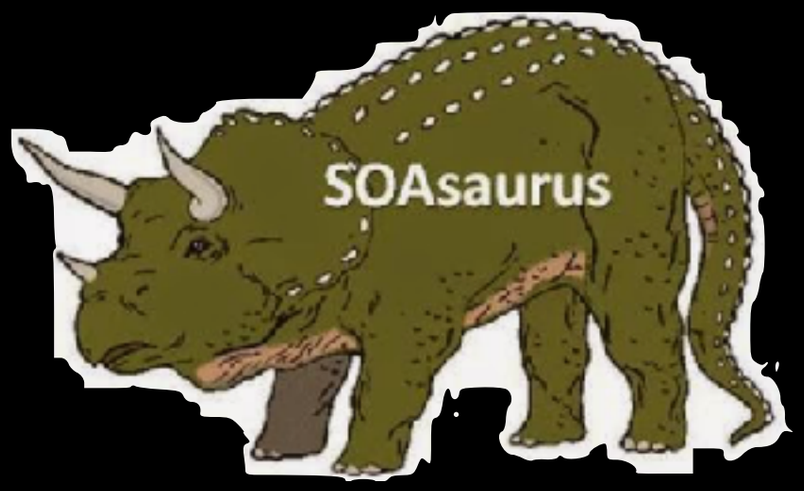
MSA 특장점

빠르고 지속적인 업그레이드는
치열한 IT비즈니스 경쟁에 있어서 가장 강력한 무기가 된다

SoA 차이점

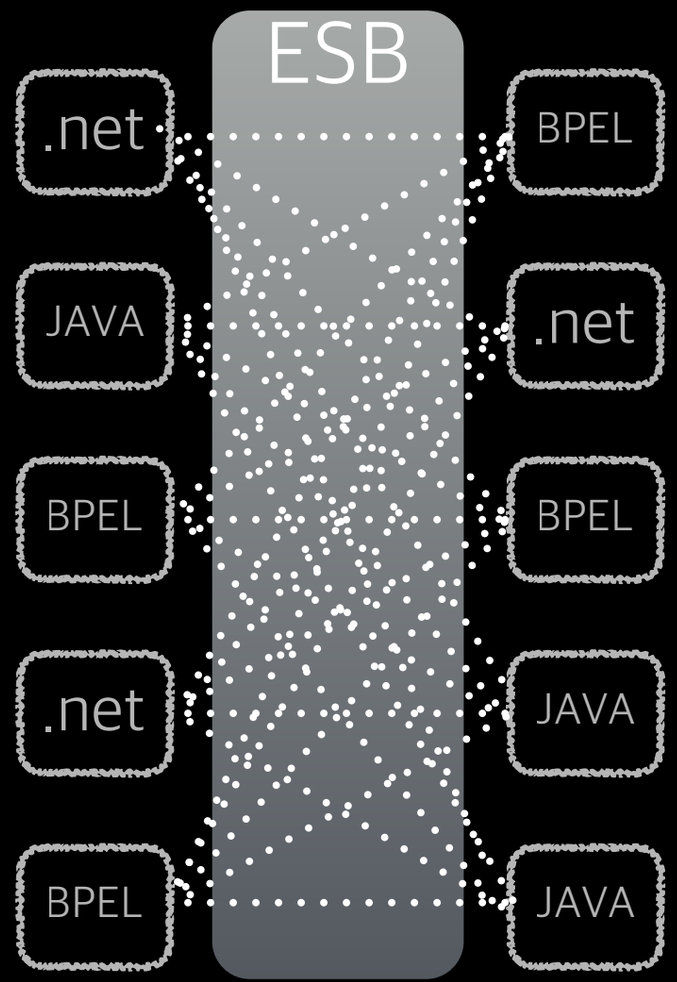
- 마이크로 서비스와 동일한 발상
- 2000년대 중반 시스템 개발의 현신적인 패러다임으로 각광
- 2009년경에 이미 부정적인 이미지가 표면화
- 중앙집중식 데이터 관리 방식 고집 (ACID 트랜젝션 유지)
- 분산 트랜젝션은 시스템 규모가 커지면 커질수록 치명적 성능 저하





ESB, API G/W

- ESB
  (Enterprise Service Bus)
- API Gateway
- DDD
  (Domain Driven Design)

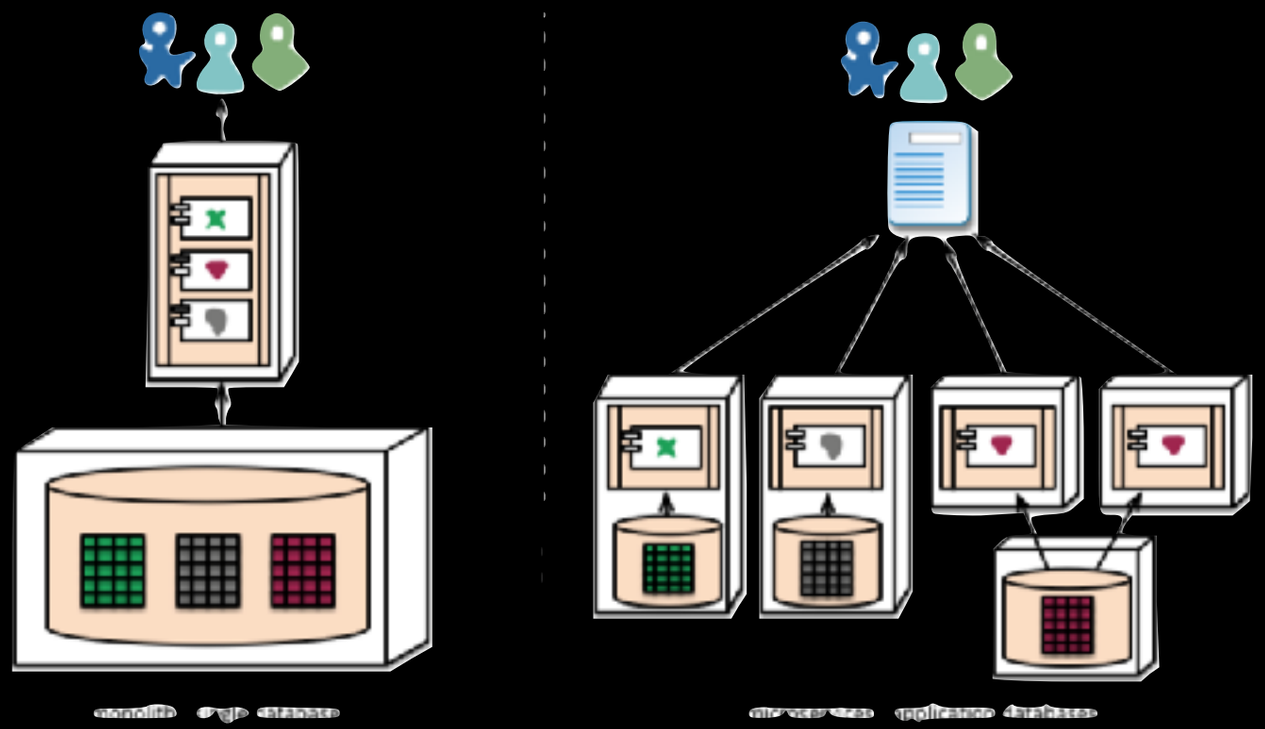


MSA 구현요소

- 데이터 분산관리
- RESTful / AscyncProcess
- 폴리그랏 아키텍쳐
- DevOps
- 클라우드 컴퓨팅
- Design for Failure

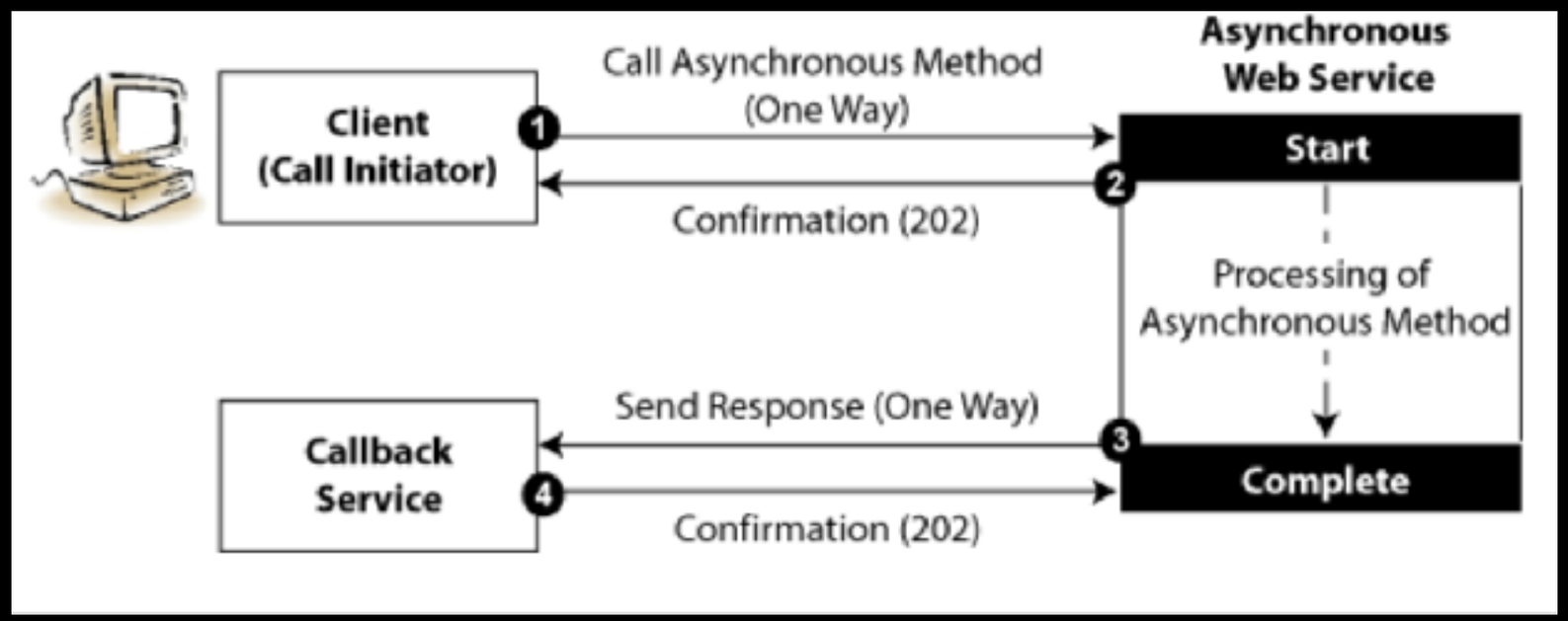
데이터 분산관리

- Not 분산 트렌젝션
- 중앙집중형 데이터 관리 방식의 반대
- 필요에 따라 다양한 DB 사용 (Polyglot Persistence)



RESTful / AscyncProcess

- 느슨한 결합과 처리의 비동기화를 위해 가장 효과적인 인터페이스
- 비동기처리는 병렬처리를 위해서 필수 불가결한 존재
- 동기화 및 트랜젝션에 대한 보장을 포기

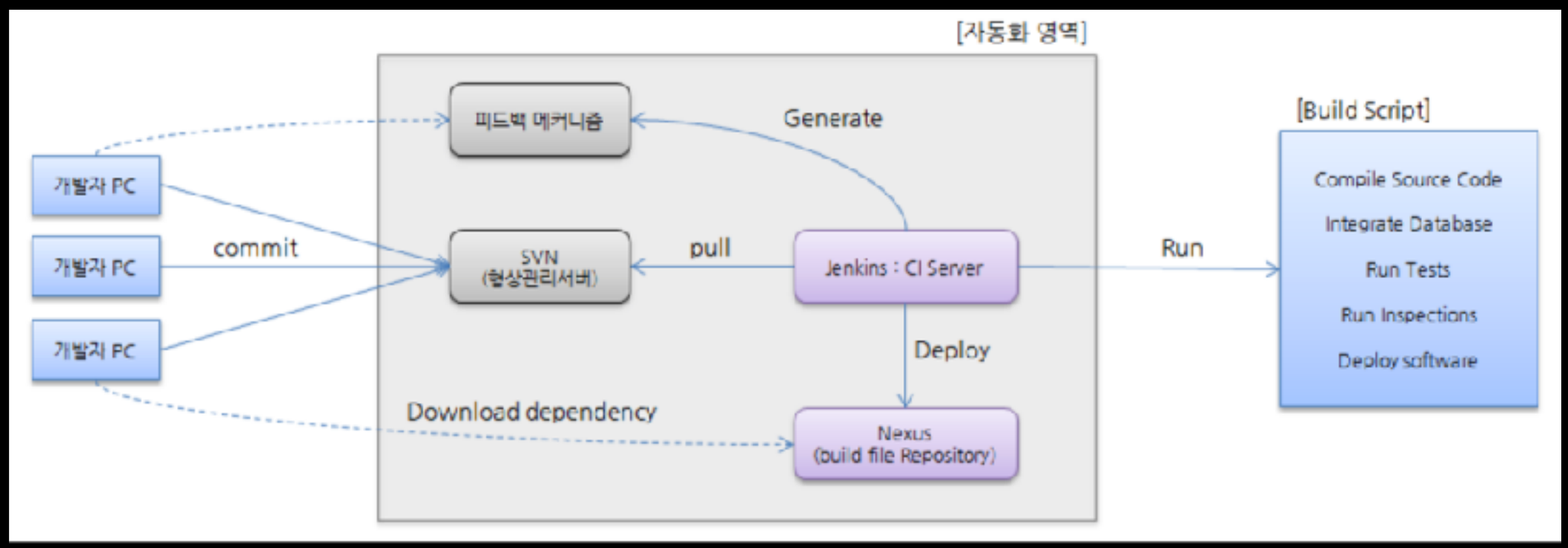


폴리그랏 아키텍쳐

- 멀티 랭귀지, 멀티 퍼시스턴스를 이용한 개발 스타일
- 각각의 서비스 목적에 맞추어 효율적인 언어와 플랫폼을 선택
- 언어나 프레임워크별로 발생하는 코드의 중복 및 관리 해야하는 기술 영역이 늘어남

DevOps

- DevOps는 CI에서 좀더 진화된 형태
- 개발, 테스트, 배포를 모두 자동화 시켜 개발 사이클이 끊임없이 순환
- 개발 속도를 최대화 시키는 개발스타일
- 서비스 연동 집합체 만큼 필요한 배포 및 테스트



클라우드 컴퓨팅

- 독립된 배포
- 각각의 서비스별로 관리 가능한 확장성
- 컨테이너 방식의 개발 + DevOps
  - 빠른 개발
  - 서비스 안정성
  - 유연한 확장성

Design for Failure

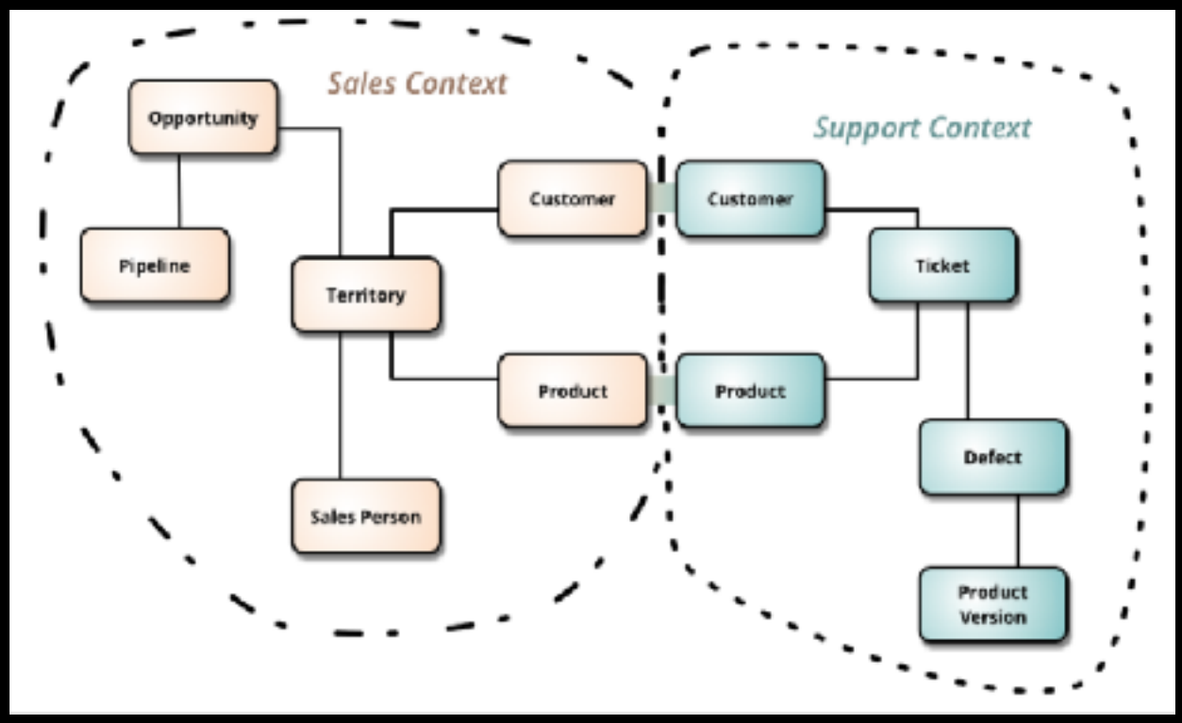
- 비동기 처리와 데이터의 분산관리
- 여러 마이크로서비스를 이용한 어떠한 복합기능이 도중에 실패 - 롤백의 어려움
- 개별 마이크로서비스 인스턴스 실패를 고려한 고 가용성 설계

문제점과 해결책

- 서비스 범위 설정 문제
- 레거시 시스템과 공존
- 운영 오버헤드
- 인터페이스 불일치
- 코드 중복
- 데이터 중복
- 분산시스템의 복잡성과 비동기성
- 테스트의 까다로움

서비스 범위 설정 문제

- 제한된 컨덱스트 (Bounded Context)
- 실제 문제 영역에서 '고객'이나 '제품'이 서로 다른 컨텍스트에 독립된 형태로 존재
- Multitenancy



레거시 시스템과 공존

- 기존 모놀리틱 시스템과 공존
- SoA 연동
- API G/W

운영 오버헤드

- 많은 양의 배포작업
- 개별적인 릴리즈
- 운영팀에서 일괄적으로 관리 불가능
- 배포작업 자동화 필요
- DevOps

인터페이스 불일치

- 서비스간의 통신 인터페이스 필요
- 인터페이스 변경에 따른 구성요소 수정 필요
- 최초 의도 외의 인터페이스 오용 가능성
- 서비스 개발 팀간의 커뮤니케이션에 좌우

코드 중복

- 다양한 언어 (Polyglot) 개발 환경에서 코드 중복은 피할 수 없음
- 가급적 단일 언어내의 동일한 프레임워크로 코드 중복 방지
- 단일 언어의 장점과 다양한 언어 개발의 실용성 이해 득실

데이터 중복

- 분산 데이터 관리 측면
- Design for failure 설계에서 중복은 피할 수 없음
- 성능과 비용의 측면에서 전체 시스템 고려

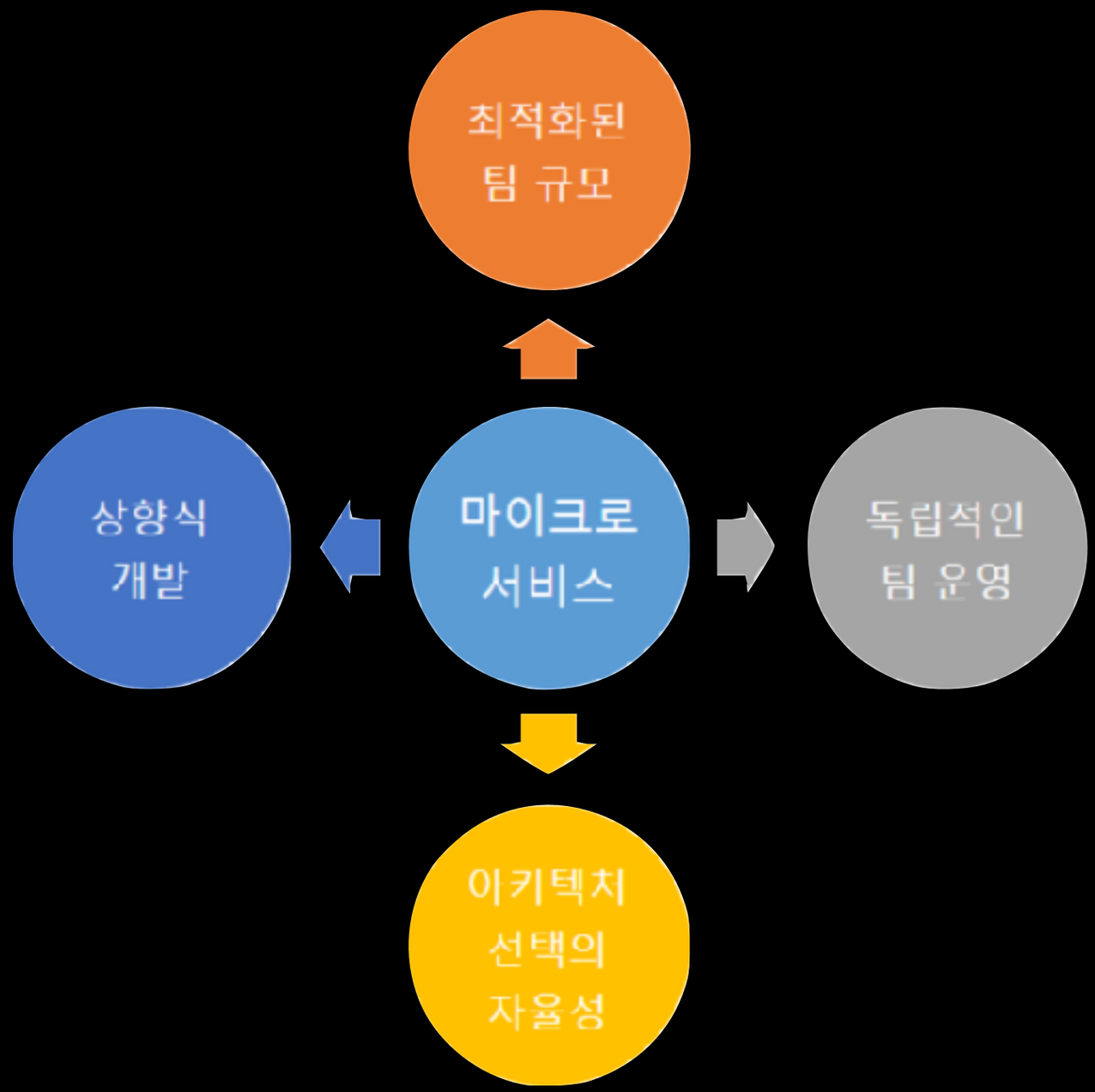
분산시스템의 복잡성과 비동기성

- 시간의 흐름에 따른 구조적 프로그램 탈피
- 마이크로 서비스 간의 관계에 따른 객체지향 프로그램 필요
- Actor Model (Reactive programming 참조)

테스트의 까다로움

- 비동기 특성에 따라 단일 테스트는 쉽지만 실제 운영 환경에서 상호테스트 어려움

마이크로서비스 개발을 위한 개발 팀 운영



최적화된 팀 규모

- 지식노동자에게 최적화된 팀 규모에 적합한 개발 규모제공
- 생산성과 효율 문제
- 작은 규모는 변화에 대한 수용과 대응에 있어서 훨씬 효율적

독립적인 팀 운영

- 팀 운영에 대한 자율성
- 작은 팀 규모는 운영에 있어서도 보다 많은 자유
- 권한과 주인의식

아키텍처 선택의 자율성

- 아키텍처 선택에 대한 권한 위임
- 개발자에 대한 신뢰의 표현
- 사용할 아키텍처를 스스로 정함
- 능력있는 개발자들의 소망

상향식 개발

- SOA실패 원인
  - 비즈니스 주도 개발
  - 하향식개발
  - 단일체 개발의 광범위한 계획과 일괄적인 정책 적용을 강제

Q&A

참고

- http://www.moreagile.net/2014/10/microservices.html?m=1
- http://www.moreagile.net/2014/04/ScrumByDDD.html
- http://bcho.tistory.com/948

감사합니다