

충남대 **SW** 아카데미

중간점검 **#1**

담임강사 하 석 재
CEO, 2HCUBE
sjha72@gmail.com

전체 일정 **(7/4~10/31)**

- 2개월 온라인강의 + 오프라인 특강
- 2개월 프로젝트
 - 매주 조별 미팅(담임강사와)
 - 미리 주제에 대한 의견 조율 가능

과정

- 예비과정
 - 파이썬
 - 자바

현재상황(**AS-IS**)

- 지난 한 달 간의 일정 진행 중간점검
 - 내용 요약 및 보완

현재상황(**AS-IS**)

- 컴퓨터전공자가 알아야 하는 언어 Big3
 - 자바
 - 자바스크립트
 - 파이썬

현재상황(**AS-IS**)

- 컴퓨터전공자가 알아야 하는 언어 Big3 및 주활용 용도
 - 자바 -> 백엔드, 프론트엔드(안드로이드)
 - 자바스크립트 -> 프론트엔드, 백엔드(Node.js)
 - 파이썬 -> 딥러닝(AI), 데이터사이언스, 자동화(Automation)

백엔드과정(난이도 *****, 활용도 ***)

- 자바심화+DBMS
 - 오픈소스
- 객체지향 + MVC + IoC/DI
- 스프링 프레임워크 + 스프링 MVC
 - 스프링 삼각형
- 스프링 부트+스프링 JPA+스프링 시큐리티

클라우드과정(난이도 ***, 활용도 *****)

- 리눅스
- TCP/IP
- 도커/쿠버네티스

AI과정(난이도 ***, 활용도 ****)

- 파이썬 + 데이터사이언스
- 자바스크립트 / 마이크로프레임워크(Flask/Django)
- 웹스크래핑(HTTP+HTML DOM+BeautifulSoup + Selenium)
- EDA(Exploratory Data Analysis:탐색적 데이터 분석)
- 텐서플로2(신경망/회귀/분류/클러스터링/추천)
- DBMS + NoSQL + 빅데이터

현재상황(AS-IS)

- 각 과정간의 차이 해소 및 개발자가 이해해야 할 사항
 - 리눅스는 클라우드만? GitHub는 백엔드만?
 - 데이터베이스는 공통임
 - NoSQL이나 빅데이터를 백엔드나 클라우드에서 같이 사용
 - 백엔드과정의 데이터베이스 설치를 도커로
 - 파이썬은 객체지향언어임 -> 텐서플로/파이토치
 - 스프링의 IoC/DI개념은 프론트엔드인 Angular.js에서도 핵심사항
 - 네트워크없이 인터넷(서버,클라이언트, 모바일 등)이 돌아가나?
 - 대다수의 서버는 리눅스임

하나만 해서 먹고 살수
있나요?



하나만 해서 먹고 살수 있나요?



Yes !!! 하지만 아주 잘 해야 함
대부분의 경우는 다양한 기술 적용함
기술 사이클이 빠르고 경쟁이 심함
트렌드 이해가 필수

2022-CNU-SW-Academy-교육과정

		충남대학교	충남대학교	충남대학교
시작일~	주차별	백엔드 SW 개발자 과정	클라우드 DevOps 엔지니어	AI + Data 과정
	0	기초프로그래밍교육 + 코딩테스트 (예비교육과정)		
7월4일(월)	1	프레임워크를 위한 JAVA 심화	리눅스	프로그래밍 기초 (Numpy, Pandas)
7월11일(월)	2	실리콘밸리에서 날아온 Database	리눅스 커널, 컨테이너, DevOps	데이터 수집 및 관리
7월18일(월)	3	스프링부트 Basic (1)	스프링부트 Basic (1)	웹서비스 이론 및 구축 실습 (React 웹프로그래밍)
7월25일(월)	4	스프링부트 Basic (2)	AWS, Kafka	웹서비스 이론 및 구축 실습 (서버 프로그래밍)
8월1일(월)	5	스프링부트 Basic (3)	Kubernetes	비정형 데이터 활용 분석 프로젝트
8월8일(월)	6	Spring Data JPA	Kubernetes, Docker	머신러닝 이론 및 실습
8월15일(월)	7	Spring Security (1)	Kubernetes, Docker	딥러닝 이론 및 실습
8월22일(월)	8	Spring Security (2)	Kubernetes, Docker	추천시스템 이론 및 실습
8월29일(월)	9	(개인프로젝트) 2주	(개인프로젝트) 2주	(개인프로젝트) 2주
8월12일(월)	10	(개인프로젝트1) 상품관리 RESTful API	(개인프로젝트1) AWS Scaleout	(개인프로젝트1) AI 서비스 설계 구현
8월19일(월)	11	(개인프로젝트2) 상품관리 RESTful API	(개인프로젝트2) GCP Scaleout	(개인프로젝트2) AI 데이터 ETL 설계 구현
8월26일(월)	12	(팀프로젝트) 6주	(팀프로젝트) 6주	(팀프로젝트6주)NLP를 이용한 음식점 추천 서비스
10월3일(월)	13	(주제) 자율 RESTful API 구현 프로젝트	백엔드+AI팀 협업 프로젝트1	(팀프로젝트6주)자동 카테고리 생성을 통한 한국어 리뷰 분석 시스템
10월10일(월)	14	• 예1) 무신사 API 클론 프로젝트	백엔드+AI팀 협업 프로젝트2	(팀프로젝트6주)주가 변동성 예측 추천 시스템
10월17일(월)	15	• 예2) 인터파크 API 클론 프로젝트	백엔드+AI팀 협업 프로젝트3	(팀프로젝트6주)음악 편식기 추천 시스템-비선호 음악 배제 추천
10월24일(월)	16	• 예3) 배달의민족 API 클론 프로젝트	백엔드+AI팀 협업 프로젝트4	(팀프로젝트6주)나쁜말 요청-악플 탐지

2022-CNU-SW-Academy-교육과정

		충남대학교	충남대학교	충남대학교
시작일~	주차별	백엔드 SW 개발자 과정	클라우드 DevOps 엔지니어	AI + Data 과정
	0	기초프로그래밍교육 + 코딩테스트 (예비교육과정)		
7월4일(월)	1	프레임워크를 위한 JAVA 심화	리눅스	프로그래밍 기초 (Numpy, Pandas)
7월11일(월)	2	실리콘밸리에서 날아온 Database	리눅스 커널, 컨테이너, DevOps	데이터 수집 및 관리
7월18일(월)	3	스프링부트 Basic (1)	스프링부트 Basic (1)	웹서비스 이론 및 구축 실습 (React 웹프로그래밍)
7월25일(월)	4	스프링부트 Basic (2)	AWS, Kafka	웹서비스 이론 및 구축 실습 (서버 프로그래밍)
8월1일(월)	5	스프링부트 Basic (3)	Kubernetes	비정형 데이터 활용 분석 프로젝트
8월8일(월)	6	Spring Data JPA	Kubernetes, Docker	머신러닝 이론 및 실습
8월15일(월)	7	Spring Security (1)	Kubernetes, Docker	딥러닝 이론 및 실습
8월22일(월)	8	Spring Security (2)	Kubernetes, Docker	추천시스템 이론 및 실습

백엔드 커리큘럼

- 1주차 프레임워크를 위한 Java
- 2주차 실리콘밸리에서 날아온 데이터베이스
- 3주차 SpringBasic Part1
- 4주차 SpringBasic Part2
- 5주차 SpringBasic Part3
- 6주차 SpringJPA Part
- 7주차 SpringSecurity Part1
- 8주차 SpringSecurity Part2

왜 백엔드를 해야하나?

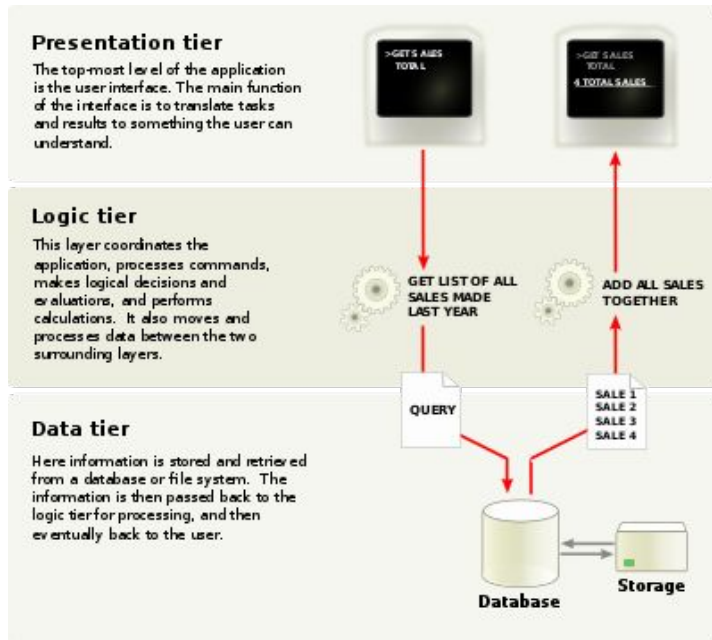
- 전자정부 프레임워크
 - 행안부(행전안전부)의 NIA(정보화진흥원->한국지능정보사회진흥원)
 - 주로 자바기반의 스프링 프레임워크 기반
 - 일정규모 이상의 프로젝트에서 사용의무화(법제화)
- 전자정부 프레임워크
 - 대학교 두 군데에서 교육

백엔드 과정을 통해서 이해해야 하는 개념

- 시스템/서비스 구분
 - Front-end / Back-end
 - Back-end
 - Presentation Layer / Business Logic / Business Object
 - 1-tier / 2-tier / 3-tier / N-tier
 - Middleware / WAS / Container / Framework
- 네트워크 프로그래밍
 - 소켓프로그래밍기반 vs. RPC/RMI
 - Sun RPC vs. DCE RPC

3계층 구조

- 표현계층
 - 주로 웹/모바일
- 비즈니스로직 계층
 - 실제 작업처리를 담당
- 데이터계층
 - DBMS와의 연결담당
 - ORM
 - JPA(Hibernate)
 - SQLMapper(MyBatis)



1-tier / 2-tier / 3-tier / N-tier

- 1-tier
 - Web Browser(Client) - Apache WebServer(Server)
- 2-tier
 - Web Browser - Web Server/DB client - DB Server
- 3-tier
 - Web Browser - Web Server/WAS client-WAS Server/DB client - DB Server
 - 트랜잭션/로드밸런싱, 장애대응, 로그취합, ...
- N-tier

객체지향

- 배경
 - 소프트웨어위기(software crisis)
 - 디버깅의 비중이 기하급수적으로 증가
- 객체지향
 - 캡슐화(encapsulation)
 - 전역변수 private으로 정의
 - getter/setter를 통해서만 접근
 - 코드 재사용(inheritance) -> MS/구글 O, 코드 구할 수 없다면 X
 - 코드가 공개가 안되면서도 재활용 가능한 방법은?
 - 컴포넌트(component) - 스프링 -> MSA(Micro Service Architecutre)

백엔드 과정을 통해서 이해해야 하는 개념

- 객체지향(Object-oriented) 기술
 - 클래스/인터페이스 -> 타입(Type)
 - 메소드, 필드
 - 상속(extends/implements)
 - 단일상속 vs. 다중상속
 - 오버로딩/오버라이딩/추상클래스
 - Subclassing/Superclassing
 - DDD(Domain-driven Design) / TDD(Test-driven Development)
 - MVC(Model-View-Controller)
 - MVVM(Model-View-ViewModel) -> SpringMVC기반

백엔드 과정을 통해서 이해해야 하는 개념

- 객체지향(Object-oriented) 기술
 - 스프링 삼각형
 - POJO(Plain Old Java Object) = IoC/DI+AOP+PSA
 - **IoC/DI(Dependency Injection)/DL(Dependency Lookup)**
 - 인터페이스기반 전략패턴사용
 - DI기반의 기능추가(결정)
 - AOP(Aspect Oriented Programming)
 - 코딩없이 이름만으로 기능추가하는 방법
 - Bytecode weaving, ...
 - PSA(Portable Service Abstraction)
 - DI로 안되어 있는 부분도 모두 DI포장을 가지도록 하는 개념(!)

백엔드 과정을 통해서 이해해야 하는 개념

- (분산)컴포넌트 기술
 - 바이너리기반의 호환성 보장
 - IDL(Interface Definition Language)
 - MIDL(Microsoft IDL), Java IDL, Reflection(*)
 - OMG CORBA
 - MS COM(ActiveX, VBX/COX)/COM+/DCOM
 - Java Beans, Enterprise JavaBeans, Spring Bean
- SOA(Service Oriented Architecture) -> MSA(Micro Service Architecture)
 - ESB -> API Gateway
 - XML -> JSON/YAML

Front-end / Back-end

- 눈에 보이는 프론트엔드와 안보이는 백엔드인 2단계로 구분
 - IBM기반의 금융시스템에서 유래
- 현재는 Presentation Layer / Business Logic layer / Business Object layer 의 3단계로 구분

Spring, SpringMVC, SpringBoot

- 스프링 프레임워크
 - 비즈니스 로직을 프로그래밍하는 분산컴포넌트 기반의 프레임워크
 - DI(Dependency Injection)기반 프레임워크
- 스프링 MVC
 - 표현계층(Presentation Layer)을 담당하는 기술 cf. Struts 1 / 2
 - SpEL(Spring Expression Language)를 사용해서 렌더링(rendering)
 - 표현계층에서 최대한 로직관련 코드제거
 - 프론트 컨트롤러 패턴 적용한 MVVM(Model/View/ViewModel) 사용
- 스프링부트
 - 스프링 프레임워크에서 가장 어려운 설정/DI관련 자동화적용해서 사용하기 쉽도록 개선

SpringJPA(Java Persistence API)

- 비즈니스 오브젝트를 담당하는 프레임워크
- ORM(Hibernate) 와 SQLMapper(MyBatis)
 - ORM이 JPA로 됨

컴포넌트



객체지향 기술의 발전(1990-> 2022)

OOP -> 컴포넌트(Component) -> “분산” 컴포넌트(스프링)

서비스기반 아키텍처 (SOA:Service Oriented Architecture) ->

마이크로 서비스 아키텍처 (MSA:Micro-service Architecture) -> 스프링 MSA

OOP -> MVC(Model/View/Controller) 아키텍처 -> DI(Dependency Injection)

객체지향의 상속은

- 상속은 소스레벨 재사용성
 - 하지만 소스가 있어야만 가능한 한계
- 컴포넌트
 - 컴파일된(바이너리) 레벨의 재활용기술 등장
 - 프로퍼티(Property)를 통한 호출 및 제어가능
 - Reflection- 자바
 - Interface Definition Language(IDL) - MS IDL(MIDL)

분산컴포넌트기술

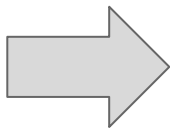
- RMI/RPC 기술을 응용해 컴포넌트를 분산아키텍처에서 사용하도록 만듦
- CORBA(IIOP) 가 원조
- Java EJB(Enterprise Java Beans) / MS DCOM(Distributed COM)
- 현재는 **Spring Bean**이 주류

컴포넌트

- CORBA(Common Object Request Broker Architecture)
- MS
 - COM(Component Object Model) / COM+ / DCOM(Distributed COM)
 - ActiveX
- 자바
 - 자바빈(Java Beans)
 - Enterprise Java Beans(EJB)
 - **Spring Bean**

분산컴포넌트기술

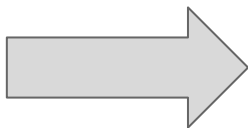
- RMI/RPC 기술을 응용해 컴포넌트를 분산아키텍처에서 사용하도록 만듦
- CORBA(IIOP) 가 원조
- Java EJB(Enterprise Java Beans) / MS DCOM(Distributed COM)
- 현재는 **Spring Bean**이 주류



SOA/MSA로 발전

SOA

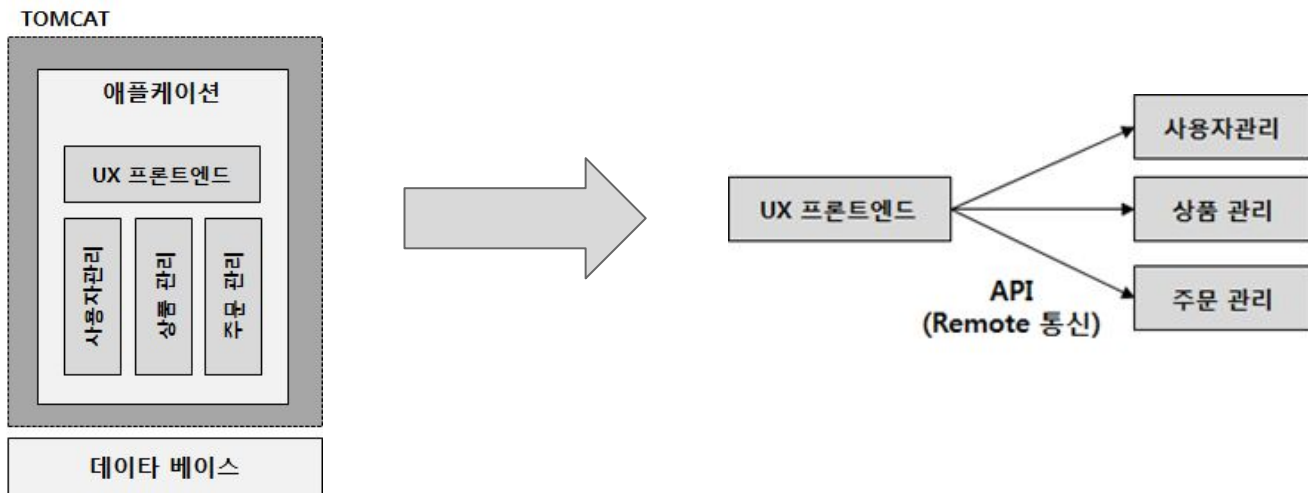
- 서비스기반 구조
- Service-oriented Architecture
- 모든 것을 웹서비스(WebService)로 구현
- WS-*로 되어있는 프로토콜의 집합
 - 주로 XML로 되어 있고 SOAP(Simple Object Access Protocol)기반
 - 무거운 서비스로 인해 JSON과 RESTful로 대체되어 발전
- ESB(Enterprise Service Bus) 때문에 무거워짐



복잡하고 무거워서 실패

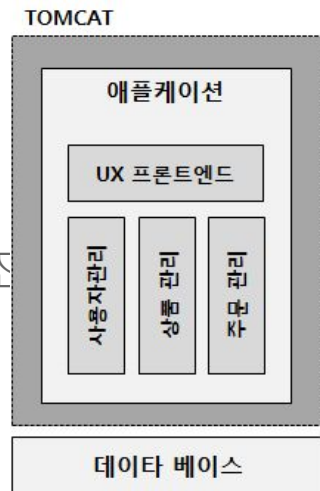
MSA

- Micro-service Architecture
- SOA를 가볍게 만든 구조
 - 기존의 모노리틱 구조를 개선



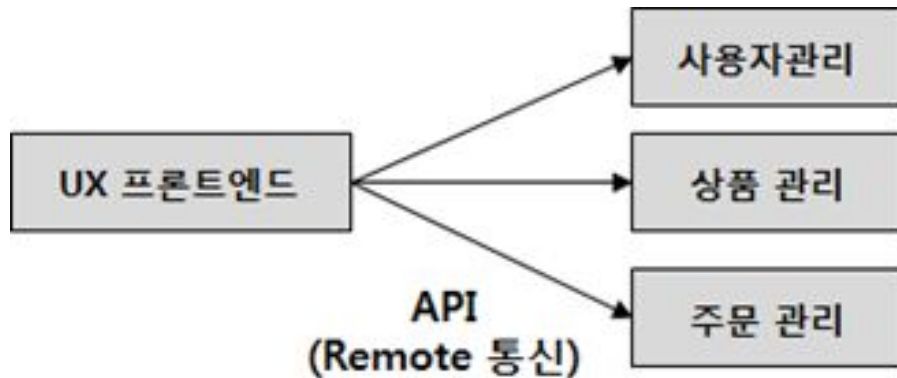
MSA

- <http://bcho.tistory.com/948>
- 모노리틱 아키텍처(Monolithic Architecture)
 - Tightly Coupled
 - 전체 시스템을 파악해야 함
 - 작은 변화가 전체 시스템에 영향을 주는 구조
 - 작은 규모의 프로그래밍에 적합



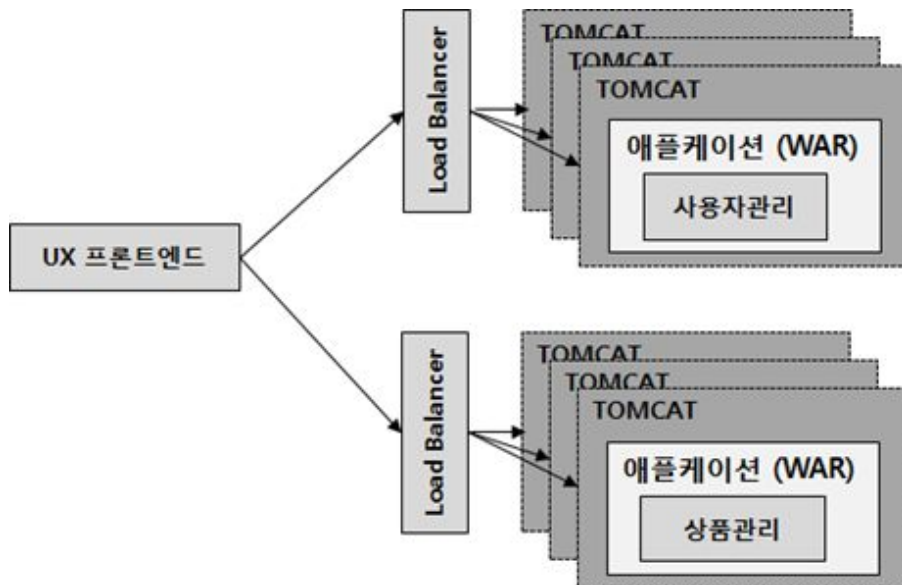
MSA

- MSA:Micro Service Architecture
 - SOA(Service Oriented Architecture)에 기반
 - 컴포넌트는 서비스의 형태로 구현
 - API를 이용해 타 서비스와 연동
 - Loosely coupled



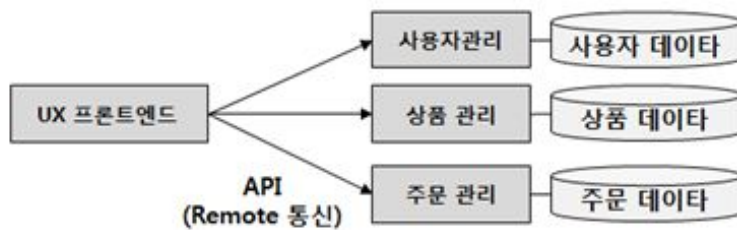
MSA

- 애플리케이션 로직을 분리, 여러 개의 애플리케이션으로 마이크로 서비스로 만들고 서비스로 별로 로드밸런서를 연결하는 방식





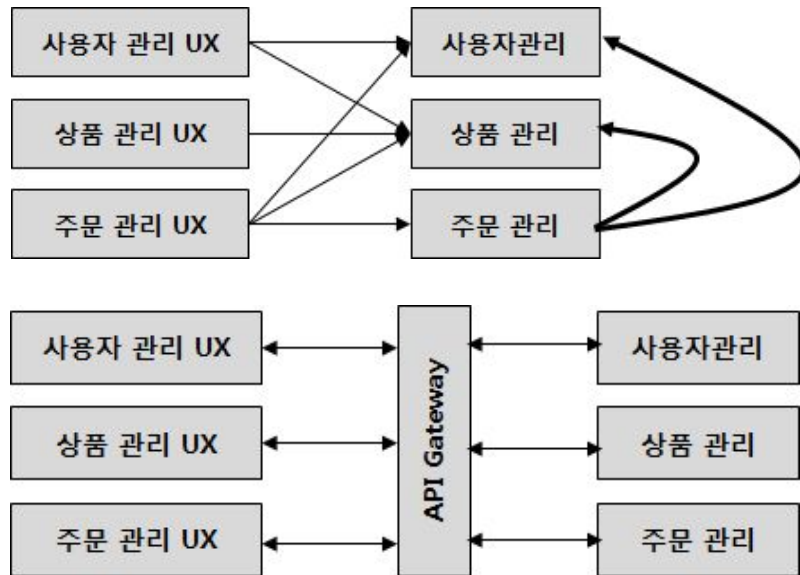
모노리틱 구조에서
데이터 저장 방식



마이크로 서비스 아키텍처에서
데이터 저장 방식

- API Gateway

- SOA의 ESB(Enterprise Service Bus)의 경량화 버전
- 미들웨어



MSA의 장점

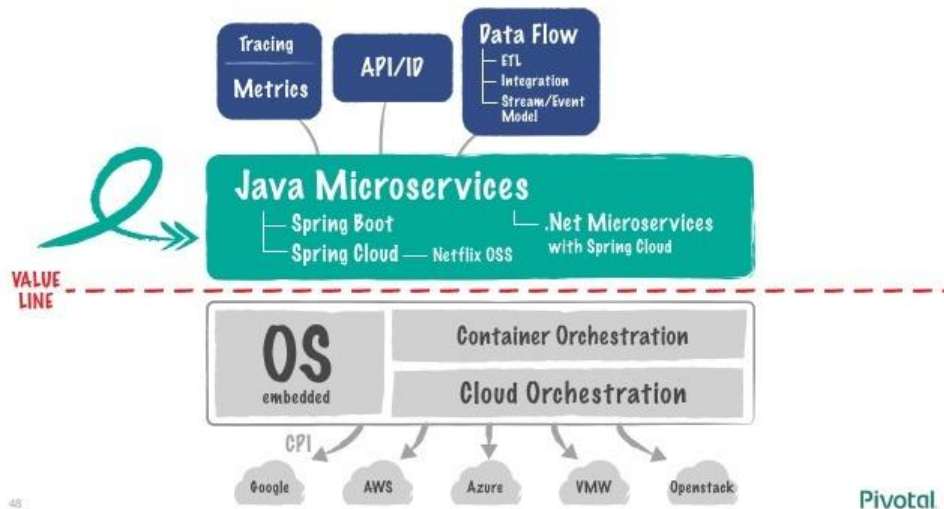
- 배포
 - 필요한 부분만 부분 배포 가능
- 확장성
 - 부하가 많은 서비스만 확장 가능
- 컴포넌트별로 팀을 독립적으로 운영가능
 - 컨웨이의 법칙(Conway's Law)
 - “소프트웨어의 구조는 그 소프트웨어를 만드는 조직의 구조와 일치한다”

MSA의 단점

- 성능
 - 마샬링(Marshalling) 오버헤드
 - XML/JSON <-> Java Object
 - 네트워크 부하 -> 시간 지연
- 메모리
 - 메모리 효율성이 떨어짐
- 테스트가 어렵다
- 서비스간 트랜잭션 처리
 - 커밋과 롤백처리
 - XA(eXtended Architecture) 분산트랜잭션 필요
- 분산형 거버넌스(Governance)
 - 시스템을 개발하는 조직의 구조와 프로세스가 변화해야 함

MSA와 컨테이너 오케스트레이션/PaaS

- 오케스트레이션과 좋은 궁합



MSA 현재상황

- 구현기술
 - JSON-RPC(text기반) -> gRPC(binary기반)
- 트랜잭션
 - Local -> Global(분산) Transaction
 - Commit/Rollback -> 2 Phase Commit(2PC)
- API 게이트웨이
 - Kong -> **Netflix OSS(zuul)**
 - 배달의 민족, 쿠팡, ...
- 스프링 클라우드 vs. Kubernetes
 - https://zetawiki.com/wiki/Spring_Cloud와_Kubernetes_기술스택

객체지향



객체지향

- 전체코드가 클래스 하나인 경우? -> 객체지향(O)
- 디자인패턴(Best Practice-최선책)
 - MVC(Model-View-Controller) 패턴
 - 모바일/PC 동시 대응
 - MVVM(Model-View-ViewModel)
 - DI(Dependency Injection) 패턴
 - 전략(Strategy) 패턴
 - 인터페이스기반 패턴

POJO(Plain Old Java Object)

- 자바객체지향의 특징 및 정신을 요약
 - 클래스/인터페이스로 구현할 수 있는 방법을 모두 제공
 - 클래스 아닌 인터페이스를 사용을 선호
- 특정한 기술/규약에 의존 배격

객체지향

- 상속/캡슐화
- POJO(궁극적인 객체지향)
 - 특정 클래스를 상속받아야 함(X)
 - 자바는 단일상속이라 제약발생
 - Thread 생성 -> Thread 클래스 상속/Runnable 인터페이스 상속
- 스프링 삼각형
 - POJO = DI+AOP+PSA

객체지향 분석설계(**OOAD**)

- 객체지향(OO) 기술 = OOA+OOD+OOP
- 분석설계의 5원칙
 - SOLID

SOLID(객체지향설계 5원칙)

- 단일책임의 원칙(SRP:Single Responsibility Principle)
 - 하나의 클래스는 하나의 기능만
- 개방-폐쇄의 원칙(OCF:Open-Closed Principle)
 - 확장에는 열려있고 수정에는 닫혀있어야
- 리스코프 치환 원칙(Liskov Substitution Principle)
- 인터페이스 분리 원칙(Interface Segregation Principle)
- 의존성 역전의 원칙(Dependency Inversion Principle)

객체지향기술의 심화(**POJO**)

- Plain Old Java Object
- 객체지향의 특징 및 정신을 요약
- **특정 기술/규약이나 특정 클래스에 의존하는 것은 안 됨**
- 단적으로 특정 기능을 구현하기 위해서 특정 클래스를 상속받는 것은 안 됨
 - 예. 자바에서 스레드(Thread)를 생성하는 방법
 - Thread vs. Runnable
 - 선택권 보장
 - 단일상속의 문제점을 보완하는 방향으로 발전

이를 위해서 **DI + AOP + PSA**가 필요

스프링 삼각형



객체지향기술의 심화(**POJO**)

- 클래스/인터페이스로 구현할 수 있는 방법을 모두 제공
 - 클래스 아닌 인터페이스를 사용을 선호
- POJO의 열렬한 지지자
 - Spring Framework / MyBatis / ...

Interface vs. Abstract Class



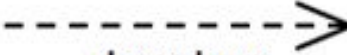





- 사이비(似而非)
 - 엄밀히는 다르지만 실제로는 비슷하게 사용
- 템플릿 메소드 패턴(추상클래스) vs. 전략패턴(인터페이스)
 - 자바의 단일상속을 고려하면 인터페이스를 사용하는 것을 선호

인터페이스를 사용하는 전략패턴(DI)이 주류

제어의 역전(**IoC**)

- Inversion of Control
- 프레임워크/WAS/미들웨어등이 주로 사용
- 제어의 권한을 넘기고 필요한 기능(메소드/함수)만 구현하는 형태
- 프레임워크는 정해진 (콜백)메소드를 호출하면 사용자의 코드가 호출되는 구조

UML기법

		
>	 dependency	 association  aggregation  composition
▷	 realization	 generalization

의존성 주입(DI)

- Dependency Injection
- 디자인패턴의 전략패턴(Strategy Pattern)
- 필요한 의존성 오브젝트를 정해진 시점에 공급
- 의존성이 없는(최소화한) 프로그래밍을 작성하라
 - 프로그램 실행단계에서 결정

의존성(**Dependency**)이란

- 어떤 프로그램이나 서비스가 수행되기 위해 필요한 것
 - 보통 리소스(자원)에 의존
- 의존성의 종류
 - 대부분 “사용(use)”
- 의존성에 방향이 있음
 - 전체는 부분에게 의존한다

프로그램에서 **DBMS**로 오라클을 사용한다

-> 프로그램은 오라클에 의존한다

객체지향기술의 심화 - 의존성주입(**IoC/DI**)

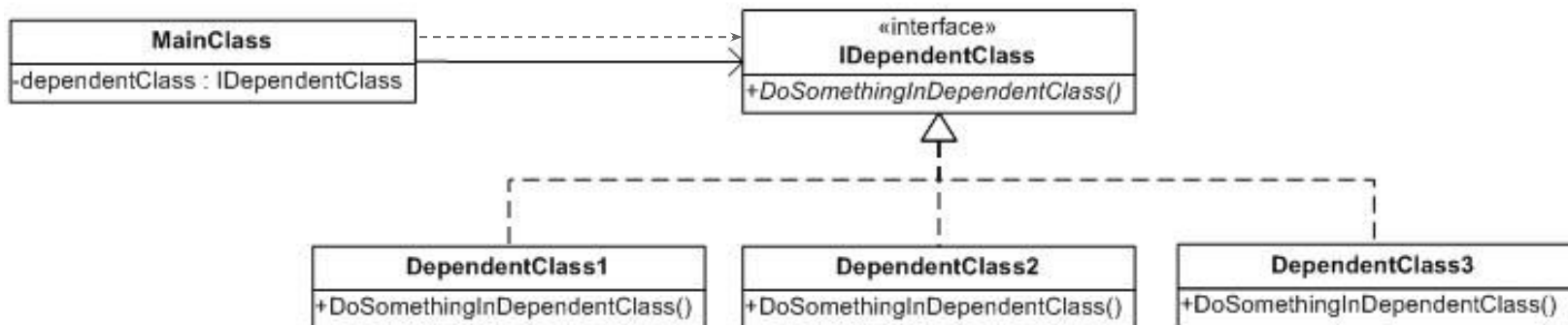
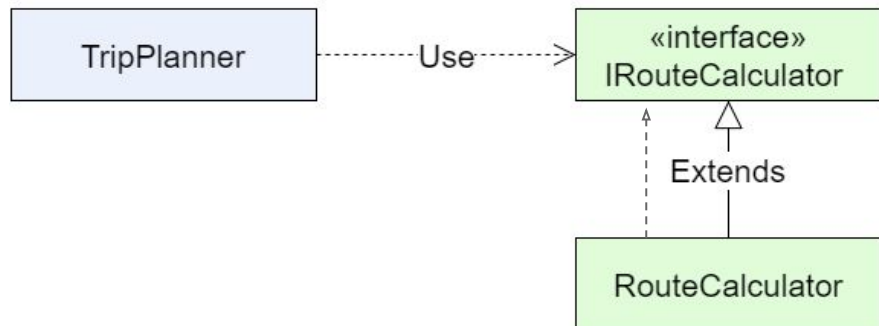
- 인터페이스에만 의존
 - 코드를 의존성이 없는 방식으로 구현
 - 오라클에서도 수행되고 MySQL도 되는 프로그램을 만들어야 한다
 - 인터페이스를 구현한 오브젝트를 계속 만들 수 있음
 - 코드의 기능추가가 매우 쉬운 구조
 - **SOLID의 OCP(개방-폐쇄) 원칙 준수**
 - 확장에는 열려있고(open) 수정에는 닫혀(closed) 있어야 한다

오라클/MySQL/SQL서버에서도 동작하는 프로그램을 짜라

객체지향기술의 심화 - 의존성주입(**IoC/DI**)

- 어떤 오브젝트를 생성할 때 필요한 구체적인 오브젝트를 지정하는 형태
- 주로 생성자를 사용해서 주입
 - setter/일반 메소드를 사용하는 경우도 있다

의존성 주입 (DI)



DI 실습

```
interface Seller {  
    public void sell();  
}  
class CupSeller implements Seller {  
    @Override  
    public void sell() {  
        System.out.println("컵을 팔아요.");  
    }  
}  
class PhoneSeller implements Seller {  
    @Override  
    public void sell() {  
        System.out.println("전화기를 팔아요.");  
    }  
}
```

```
class Mart {  
    private Seller seller;  
    public Mart(Seller seller) {  
        this.seller = seller; }  
    public void order(){  
        seller.sell(); }  
}  
class Test {  
    public static void main(String[] args){  
        Seller cupSeller = new CupSeller();  
        Seller phoneSeller = new PhoneSeller();  
        Mart mart1 = new Mart(cupSeller);  
        mart1.order();  
        Mart mart2 = new Mart(phoneSeller);  
        mart2.order();  
    }  
}
```

인터페이스 선호하는 이유(상속의 문제점)

- 클래스 상속하고 재사용 안되는 코드
 - 전체를 다 상속받아 사용안되는 코드는 군살
 - 다중 상속은 활용되지 않는 코드가 늘어날 가능성이 높음
- 단일상속만 허용
- 인터페이스는 오버헤드없는 상속이 가능
 - 코드의 통일성 유지 및 기능추가에 유연한 구조