Restful API, MSA, DevOps
조사 보고서

ICT 인턴십
상명대학교 컴퓨터과학과
최미리

목차

1. Restful API

1. Restful api의 정의

1.1 Rest의 정의

"*RestFul API(Representational State Transfer 아키텍쳐를 따르는 api)*"

- 웹이 존재하는 모든 자원(이미지, 동영상, db자원)에 고유한 uri를 부여해 활용하는 것으로 자원을 정의하고 자원에 대한 주소를 지정하는 방법론을 의미한다.

- http, uri로 잘 표현된 리소스에 대한 행위를 http method(post,get,put,delete)를 통해 정의한다. 리소스의 내용은 json, xml 등의 다양한 표현 언어로 정의된다.

- 웹 아키텍쳐의 우수성을 최대한 활용할 수 있도록 만들어진 API이다.

1.2 Rest api 의 정의

rest api : '무엇을(http uri로 정의된 리소스), 어떻게 한다 (http method+payload)로 잘 정의된 api, rest 특징을 지키면서 api를 제공하는 것을 의미한다.(일종의 Code Convention 설계원칙, 가이드를 지키면서 개발을 하자는 의미)

ex) route 설정

CRUD	HTTP verbs	Route
resource들의 목록을 표시	Get	/resource
resource 하나의 내용 표시	Get	/resource/:id
resource를 생성	post	/resource
resource를 수정	put	/resource:id
resource를 삭제	delete	/resource/:id

* :id는 하나의 특정한 resource를 나타낼 수 있는 고유의 값
  ex) student를 생성하는 route는 POST/student
  id가 13인 student를 삭제하는 route는 DELETE /studnets/13

* RestFul하지 못한 예
  - CRUD 기능을 모두 POST로만 처리하는 API
  - Route에 resource, id 외 정보가 들어가는 경우(/students/updatename, student 하나의 이름만 변경하는 API의 경우 PUT students/:id/name

2. Rest의 구성

1. 자원(Resouce) - URI /users GET
2. 행위(Verb) - HTTP Method /users/{id} GET
3. 표현(Representations) /users PUT

3. REST의 주요한 목표

- 구성요소 상호작용의 규모 확장성
- 인터페이스의 범용성
- 구성요소의 독립적인 배포
- 중간적 구성요소를 이용해 응답 지연 감소, 보안을 강화, 레거시시스템을 인캡슐레이션

4. Rest의 특징(rest 아키텍처에 적용되는 6가지 제한 조건)

1. 클라이언트/서버 구조 : 일관적인 인터페이스로 분리되어야 한다.
2. Stateless : 각 요청 간 클라이언트의 콘텍스트가 서버에 저장되어서는 안된다.
3. 캐시 처리 가능 : WWW에서와 같이 클라이언트는 응답을 캐싱할 수 있어야 한다.
4. 계층화 : 클라이언트는 보통 대상 서버에 직접 연결되었는지, 또는 중간 서버를 통해 연결 되었는지를 알 수 없다. 중간 서버는 로드 밸런싱 기능이나 공유 캐시 기능을 제공함으로써 시스템 규모 확장성을 향상시키는데 유용하다.
5. Code on demand : 자바 애플릿이나 자바스크립트의 제공을 통해 서버가 클라이언트가 실행시킬 수 있는 로직을 전송하여 기능을 확장 시킬 수 있다.
6. 인터페이스 일관성 : 아키텍처를 단순화시키고 작은 단위로 분리함으로써 클라이언트 서버의 각 파트가 독립적으로 개선될 수 있도록 해준다.

5. Restful API 장점

1. OPEN API를 제공하기 쉽다.
2. 멀티플랫폼 지원 및 연동이 용이하다.
3. 원하는 타입으로 데이터를 주고받을수 있다.(XML, JSON, RSS)
4. 기존 웹 인프라를(HTTP)를 그대로 사용가능하다 (방화벽, 장비 요건 불필요)
   사용하기 쉽다
5. 세션을 사용하지 않는다. 각각의 요청에 독립적.
6. API의 요청 방식 자체로 api가 어떤식으로 동작하는지 명확히 알 수 있고 인가된 요청자들에 대해 언제 어디서나 데이터를 제공할 수 있다.

6. Restful API 단점

1. 표준이 없어서 관리가 어렵다.
2. 사용할 수 있는 메소드가 4가지 밖에 없다.
3. 분산환경에는 부적합하다.
4. http통신 모델에 대해서만 지원한다.

7. Restful한 API 설계 규칙

1. uri 설계하기

uri(Uniform Resource Identifier) : 균등한 리소스 식별자/ 인터넷의 어떠한 리소스를 식별하기 위해서 만들어진 것.

- 소문자를 사용하자(대소문자 구분한다.)

- 하이픈을 사용하자(공백 대신에)
- 확장자를 사용하지 말자(대신 Accept header)

2. crud는 uri에 사용하면 안된다.
   ex) GET /posts/13/delete HTTP/1.1

3. 컬렉션과 도큐먼트에서 단수 복수도 정확히 지켜주자.!
   ex) http://www.remotty.com/sports/soccer
   http://www.remotty.com/spoarts/soccer/players
   http://www.remotty.com/sports/soccer/players/13/skills
   여기서 http://www.remotty.com/sports는 컬렉션
   sports컬렉션에는 players라는 도큐먼트가 있는 것.

4. 반응형 웹에서의 REST : User-Agent를 이용하여 다른 곳으로 리다이렉트 시켜주는게 아니라 URI는 그대로이지만 화면만 장비에따라 알아서 최적화 되도록 설계 해야 한다.

5. 응답 상태 코드 적절히 사용
   - 200 성공
   - 400 Bad Request - field validation 실패시
   - 401 Unauthorized - API 인증,인가 실패
   - 404 Not found
   - 500 Internal Server Error - 서버 에러

6. 동사보다는 명사를 사용하자

HTTP POST : /getDogs	HTTP GET:/dogs
HTTP POST : /setDogsOwer	HTTP POST:/dogs/{puppy}/owner/{terry}
(나쁜예)	(좋은 사용예)

7. 단수보다는 복수형을 사용하자
   리소스는 복수형 명사를 사용
   ex) dog(X)
   dogs(O)
8. 많은 도큐먼트를 리턴할 때에는 잘라서 리턴하는 페이징 처리가 필요하다.
   ex) Facebook style
   :/records?offset=100&limit=25(100

9. 리소스와 행위를 명시적이고 직관적으로 분리한다.
   ex) http://서버:8080/delete/member

- uri 안에 행위 포함하는 표현 없어야 한다.

- 행위는 GET(조회),POST(삽입),PATCH(일부 수정),PUT(객체 전체 수정),DELETE(삭제),HEAD(Body없이 헤더만 받음), OPTIONS(해당 리소스에 가능한 operation을 응답) 등으로 표현한다.

10. Message는 Header와 Body를 명확하게 구분하여 분리해 사용한다.
- Entity에 대한 내용은 Body에 담아 전송한다. -> 수정하거나 삽입하려는 정보에 대한 내용은 Body에 담아 전송한다.
- API 버전 정보,MIME 타입 등은 header에 담는다.

11. 자원에 대한 행위는 method로 표현

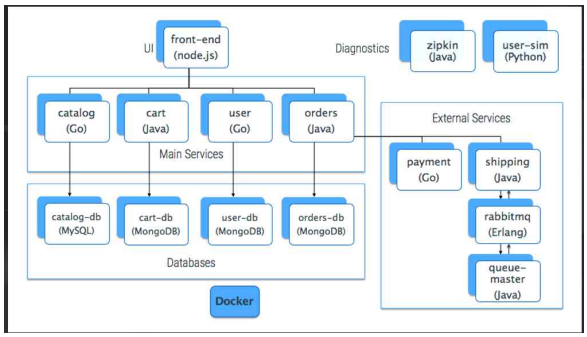
rest api에서 사용되는 method : post, get(조회), put(수정), delete(삭제), [patch]

ex) POST http://localhost:5000/user/who1sth1s
    who1sth1s 유저 생성
ex) GET http://localhost:5000/user/who1sth1s
    who1sth1s 유저의 정보를 가져옴
ex) PUT https://localhost:5000/user/who1sth1s
    who1sth1s 유저 정보 수정
ex) DELETE https://localhost:5000/user/who1sth1s
    who1sth1s 유저 정보 삭제

12. 리소스간의 관계를 표현하는 방법

/resource명/{그 리소스에 대한 identifier}/{연관되는 다른 리소스 other-related-resourcce}
ex) owner가 가지고 있는 개(dogs) 목록 : "GET /owner/{terry}/dogs"

2. 마이크로 서비스 아키텍처(MSA)



1. MSA의 정의

하나의 큰 애플리케이션을 여러 개의 작은 애플리케이션으로 쪼개어 변경과 조합이 가능하도록 만든 아키텍처를 말한다. 작은 단위로 기능을 분할할 때 수평 방향의 계층별 절단이 아니라 수직 방향의 기능별로 절단한다. 절단된 작은 모듈을 마이크로 서비스라 한다.

각 마이크로 서비스는 공유나 프로세스 간 통신이 없어도 독립적으로 실행되며 운영 관리된다. 마이크로서비스 간 연결은 응용 프로그래밍 인터페이스를 이용한다. 마이크로 서비스는 자원 표현이나 데이터 관리 등에 있어서 기능적으로 완전해야 한다.

2. 모노리틱 아키텍쳐와의 비교

2.1 모노리틱 아키텍쳐 정의

기존의 전통적인 웹 시스템 개발 스타일로 **하나의 애플리케이션 내에 모든 로직들이 들어가 있는 구조.** 각 컴포넌트들은 상호 호출을 함수를 이용한 **call-by-reference** 구조를 취한다. 소규모의 프로젝트에서는 전체 애플리케이션을 하나로 처리하기 때문에 편리하지만 대형 시스템 개발 시 몇가지 문제점이 있다.

2.2 모노리틱 아키텍처의 장점

1. 기술의 단일화
2. 관리 용이성
3. 심플한 구조
4. 간편한 코드관리
5. 간편한 트랜젝션 관리
6. 설계/테스트가 간편

2.3 모노리틱 아키텍쳐의 단점

1. 크기가 커서 빌드 및 배포시간, 서버의 기동시간이 오래걸린다.
2. 협업 개발하기가 어렵다.
3. 시스템 컴포넌트들이 call-by-reference 기반으로 연결되어 있어 전체 시스템의 구조를 제대로 파악하고 개발해야 하기 때문에 시스템 구조가 커질수록 개인이 전체 시스템의 구조와 특성을 이해하기 어렵다.
4. 컴포넌트별로 기능/비기능적 특성에 맞춰 다른 기술을 도입하고자 할 때 유연 하 지 않다.

이러한 단점으로 인해서 마이크로 서비스 아키텍처가 등장했다.
하지만 어느 것이 더 나은 개발 아키텍쳐라고 할 수 없다. 각 프로젝트의 상황이나 성격에 따라 알맞은 아키텍처의 선택이 필요하다.

3. MSA의 등장배경

1. 기술 환경의 변화
   - 하드웨어의 저렴화와 고속화
   - 네트워크의 고속화 및 보급
   - 클라우드 환경의 중심

2. 웹 서비스 레거시 화
   - 기업의 IT 환경이 보다 거대하고 복잡하게 되었다.
   - 서비스마다 특성과 변화 속도가 크게 다르기 때문에 표준화가 어렵다.
   - 거대한 웹 서비스를 관리하기 위해서는 필연적인 선택은 MSA이다.

3. 서비스 공유의 일반화
   - SDx 흐름에 따른 다양한 가상화 기술
   - 서비스에 대한 성능과 가용성을 보장
   - API-Gateway를 통하여 서비스의 공유화를 실현

4. SOA(Service Oriented Architecture 와의 차이

SOA는 엔터프라이즈 시스템을 중심으로 고안된 아키텍쳐라면, 마이크로 서비스 아키텍쳐는 SOA 사상에 근간을 두고 대용량 웹서비스 개발에 맞는 구조로 사상이 경량화되고, 대규모 개발팀의 조직 구조에 맞도록 변형된 아키텍쳐이다.

	서비스 지향 아키텍처(SOA)	**마이크로 서비스 아키텍처(MSA)**
범위(scope)	전사적 아키텍처	한 프로젝트에 대한 아키텍처
유연성(flexibility)	오케스트레이션에 의한 유연성	빠른 배포와 신속하고 독립적인 개발에 의한 유연성
조직(Organizati	다양한 조직 단위에 의한 서비스	동일 프로젝트 내의 팀들에 의한

on)	구현	서비스 구현
배포(Deployment)	여러 서비스들의 단일 배포	각 서비스들의 독립적인 배포
사용자 인터페이스(ui)	모든 서비스들을 위한 보편적인 UI로서의 포털	서비스가 UI를 포함

5. MSA의 특징

1. Decoupled : 서비스 간의 종속성 배제
2. Well Defined Interface : 서비스 간의 통신을 위해서 잘 정의된 API가 필요
3. Independent : 각 서비스 별로 독립적으로 개발 및 운영할 수 있다.
4. Conway's Law 적용 : 결국 서비스의 구성 디자인은 그 서비스를 구현하는 조직의 모습에 기반한다.(비지니스 기능 중심의 구성)
5. 스마트 엔드 포인트와 간단한 파이프 : 복잡한 프로토콜이 아닌 restful한 http api 및 경량 메시징 프로토콜을 사용
6. 개발의 분권화 : 특정 기술에 고착되고 중앙 집권적인 관리에서 가볍고 최적화된 기술 선택으로 전환
7. 데이터 관리의 분권화 : 개별 서비스마다 데이터베이스를 소유
8. 인프라 자동화 : 프로비저닝[1]을 통한 인프라 구성에 대한 자동화
9. 장애를 전제로 한 설계 : 장애가 서비스에 영향을 주지 않도록 하며, 모니터링 체계 구축
10. 변화에 대응하는 설계 : 변화에 쉽게 대응할 수 있도록 작은 규모의 조직 구성

6. MSA의 장점

1. 각 모듈은 독립적이기 때문에 해당 모듈에 가장 잘 맞는 프로그래밍 언어를 선택 할 수 있다.
2. 각 모듈마다 자신의 데이터베이스를 가지고 있기 때문에 제약없이 noSQL또는 관계형 데이터베이스를 선택할수 있다.
3. 개발자는 마이크로서비스의 개발과 관리함으로 모듈에 대한 방대한 지식을 갖게된다. 다기능 멤버들은 서비스를 위해 함께 일한다.
4. 서비스단위로 작게 개발되기 때문에 빌드 및 배포시간이 상대적으로 짧다.
5. 확장성이 높다. 유연한 확장 모델을 가질 수 있다.
6. 개발 및 유지보수에 드는 시간과 비용을 절감 할 수 있다.

7. MSA의 단점

1. 서버는 다른 서버를 호출할 수 있기 때문에 큰 규모의 프로젝트에서는 서비스 호출을 추적하거나 서비스들을 모니터링하기 힘들다.
2. 하나의 서비스는 다른 서비스와 REST API를 통해 소통하기 때문에 단일체의 프로세스 간 통신에 비해 느리다.

---

1) 사용자의 요구에 맞게 시스템 자원을 할당, 배치, 배포해 두었다가 필요 시 시스템을 즉시 사용할 수 있는 상태로 미리 준비해 두는 것을 말한다

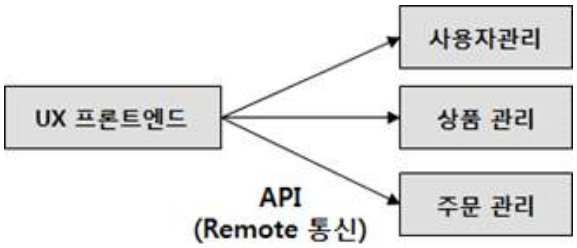
3. 디버깅이 힘들다. 서비스 호출이 다른 서비스를 연속적으로 호출하는 경우에 특히 힘들다.
4. 서비스 별로 중복되는 라이브러리나 , 여러개의 WAS로 CPU나 메모리 자원의 오버헤드도 존재한다.
5. 서비스간 트렌젝션 처리가 어렵다.

8. 마이크로 서비스 아키텍쳐의 구조

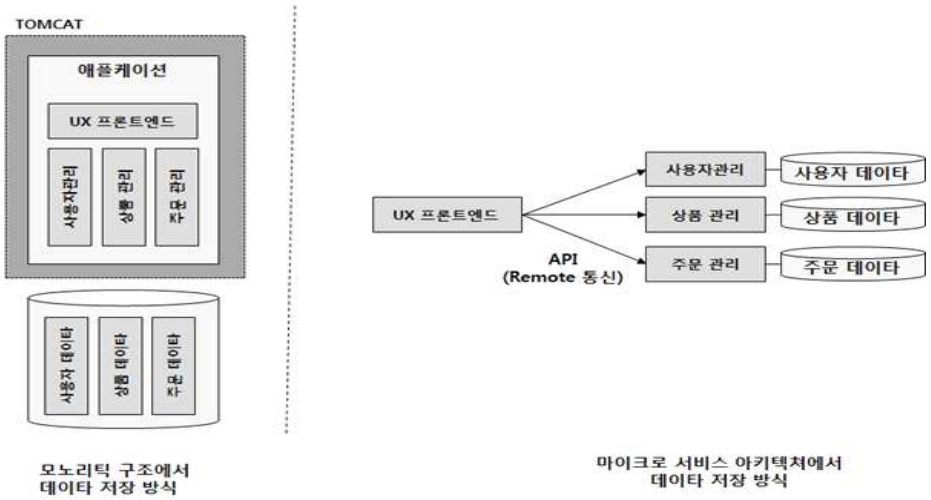
8.1 서비스

각 컴포넌트는 서비스라는 형태로 구현되고 API를 이용하여 타 서비스와 통신
각 서비스는 독립된 서버로 타 컴포넌트와의 의존성이 없이 독립적으로 배포된다.

8.2 api remote 통신



8.3 데이터 분리



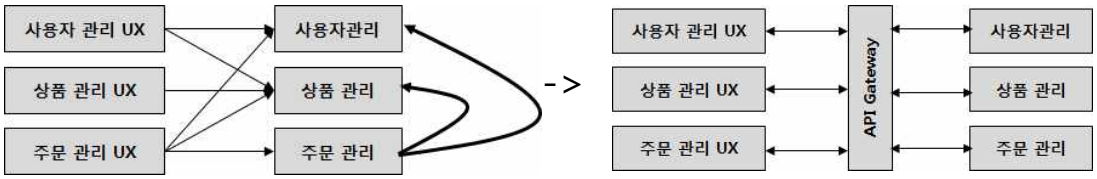
MSA 구조에서는 중앙 집중화된 하나의 통 데이나 베이스를 사용하는 것이 아니라 서비스 별로 별도의 데이터 베이스를 사용한다.

9. API Gateway(MSA의 핵심 컴포넌트)

프록시 서버처럼 api들 앞에서 모든 api에 대한 end point를 통합하고, 몇 가지 추가적

인 기능을 제공하는 미들웨어, SOA의 ESB(Enterprise Service Bus)의 경량화 버전

9.1 endpoint 통합 및 토폴로지 정리



각 서비스가 다른 서버에 분리 배포되기 때문에 API의 EndPoint(서버의 URL가 각각 다르다)-URL의 개수가 너무 많아짐.

9.2 Orchestration

여러개의 서비스를 묶어서 하나의 새로운 서비스를 만드는 개념

ex) 포인트 적립 서비스 + 물품 구매 서비스 = 물품 구입 시 포인트 적립

넷플릭스의 경우 마이크로 서비스 아키텍쳐 사용하면서 여러개의 서비스들을 gateway 계층을 통해서 orchestration 하는 모델을 사용하고 있다.

9.3 공통 기능 처리

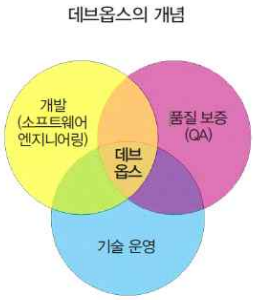
api에 대한 인증이나 logging 과 같은 공통 기능에 대해서 서비스 컴포넌트 별로 중복 개발해야 하는 비효율성 유발할 수 있다. api gateway에서 이러한 공통 기능을 처리하게 되면 api자체는 비즈니스 로직에만 집중해 개발에 있어서 중복 방지가 가능하다.

10. 마이크로서비스 설계시 고려사항

- 각 개별 서비스에 대한 기능 적합성 및 성능에 대한 효율성
- 서비스 간의 호환성
- 각 서비스에 대한 신뢰성 및 서비스 간의 통신 시의 보안
- 개별 서비스에 대한 유지보수성 및 이식성
- 개발 및 운영 복잡성
- 코드 의존성 : 개발 서비스를 독립적으로 배포 할 수 있기 때문에 다른 라이브러리 버전들이 바이너리 의존성 때문에 더 이상 호환되지 않을 수 있다.
- 성능 저하 : 서비스가 분리되어 늘어난 통신들로 인해 성능 저하될 수 있다.
- 인프라 스트럭처와 운영 : 모니터링, 배포, 테스트가 자동화되어 있어야 함

3. DevOps 데브옵스

1. DevOps의 개념



"*Development(개발) + Operations(운영)*"

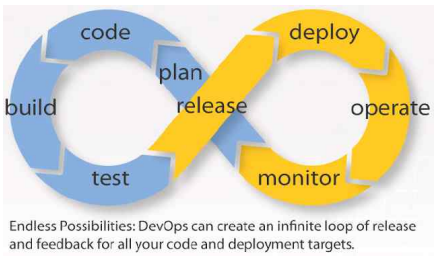
소프트웨어 개발 방법론의 하나로, 개발과 운영을 결합한 혼성어이다. 개발 담당자와 운영 담당자가 연계하여 협력하는 개발 방법론(문화)을 말한다.

2. DevOps의 목적

소프트웨어 제품과 서비스를 빠른 시간에 개발 및 배포하는 것을 목적으로 한다.
운영 프로세스의 예측 가능성, 효율성, 보안, 유지보수 가능성을 극대화함이 목적이다.

- 전반적인 타임 투 마켓
- 새로운 릴리스의 더 낮은 실패율
- 픽스 간 짧아진 리드 타임(상품 생산 시작부터 완성까지 걸리는 시간)
- 복구 시 더 빠른 평균 시간

3. DevOps의 모델



Endless Possibilities: DevOps can create an infinite loop of release and feedback for all your code and deployment targets.

- 기획/개발/테스트/유지보수의 전반적인 이슈를 공유한다.
- 클라우드/가상 환경으로 환경 제한이 현저히 줄어든다.
- 개발자/비개발자 모두가 함께 고민을 하며 함께 업무를 해결한다.

4. DevOps의 장점

- 속도 : 작업 속도가 빨라지므로 고객을 위해 빠르게 혁신하고 시장 변화에 더 잘 적응

하며 좀 더 효율적으로 비즈니스 성과를 창출할 수 있다.
- 신속한 제공 : 릴리스의 빈도와 속도를 개선해 제품을 더 빠르게 혁신하고 향상할 수 있다.
- 안정성 : 애플리케이션 업데이트와 인프라 변경의 품질을 보장한다. 모니터링과 로깅방식을 통해 실시간으로 성능에 대한 정보를 얻을 수 있다.
- 확장 가능 : 규모에 따라 인프라와 개발 프로세스를 운영 및 관리한다. 자동화와 일관성이 지원되므로 복잡한 시스템이나 변화하는 시스템을 효율적으로 관리할 수 있다.
- 협업 강화 : 개발자와 운영팀의 긴밀한 협력과 책임 공유로 워크플로를 결합해 비효율성을 줄이고 시간을 절약할 수 있다.
- 보안 : 제어를 유지하고 규정(자동화된 규정 준수 정책, 세분화된 제어 및 구성 관리 기술 사용) 준수하며 신속하게 진행할 수 있다.

5. Agile와 DevOps의 접목

agile development
- (요구사항) <--> (개발+테스팅) 사이의 거리 문제를 해결
- 고객 요구사항의 우선순위에 맞게 설계, 개발, 테스트를 교차 수행
- 기능/비기능에 초점

DevOps
- (개발+테스팅) <--> (관리+영업) 팀 사이의 거리 문제를 해결
- 자동화 릴리즈 관리
- 기능/비기능 + 관리와 영업에 초점
- 재사용성과 자동화에 집중

Agile과 DevOps의 배포 비교

agile에 비해 각 개발 생산 과정이 지속적인 빌드의 형태를 취하고 바로 생산에 반영된다. 결론적으로 Agile을 위해 DevOps가 필요. DevOps는 Agile에 종속적이다.

6. DevOps의 원칙

1. 주요 기능에 집중하고 있는가?
2. 품질을 내재화하기 위하여 노력하고 있는가?
3. 개발에 필요한 지식을 창출하기 위해서 과학적으로 접근하고 있는가?
4. 완벽한 명세서를 만들기 위한 비용보다, 명쾌한 협업을 중시하여 커뮤니케이션 비용을 지출하고 있는가?
5. 가능한 한 빨리 개발하기 위해서 시도하고 있는가?
6. 사람을 존중하는 개발자 문화를 만들고 있는가?
7. 최적화를 위한 방안을 고안하는데 회의나 토론을 아까워하지 않고 있으며, 그것에 대해서 투자를 아낌없이 하고 있는가?

7. DevOps의 적용

7.1 Devops의 적용 조건

- 개발/테스트 환경은 실제 환경과 유사하다.

- 디플로이 빌드[2]가 빈번하다
- 지속적으로 개발과정 도중의 품질을 검증한다.

7.2 Devops의 적용 분야
- e 비즈니스 및 웹 사이트 프로젝트(Amazon, Flickr, Groupon등)
- 클라우드 플랫폼(IaaS, Paas)

7.3 DevOps의 적용을 피해야 할 분야
- 중요성이 높은 응용 프로그램(금융, 전력 시스템 등)
- 높은 완결성을 필요로 하는 프로그램(패키지, 제조, 건설 시스템 등)

7.4 DevOps 툴
- puppet
- chef

7.5 DevOps의 적용 사례
1. Flickr : 사진 중심의 SNS 서비스. 거의 모든 것을 자동화하고 개발과 운영 각자만의 문화를 바꾸어 서로 협력해 하루에도 10번씩 배포할 수 있도록 하였다.
2. netflix : 오프라인 영화 렌탈 체인을 함몰시킨 VOD 회사. AWS에 10000개 이상의 인스턴스를 10명의 DevOps팀으로 운영하고 개발 지원.
3. fotopedia : 인간 중심의 사진이라는 모토로 flickr와 Wikipedia를 섞어놓은 서비스 제공. 일일 1억 5000만의 페이지뷰 보유

8. DevOps의 최근동향
- mobile : 빠른개발, 변화관리, 테스팅 자동화 방법 등
- Cloud : SaaS/PaaS/BaaS 등을 클라우드와 DevOps연계한 Agile 증대 및 서비스 가상화 기술이용
- SDN : 자동화 지원, 통합 환경 구성 시험 실증
- Security : [3]Compliance 공유, 감사영역의 보안 효율성 증대

9. DevOps의 미래

"DevSecOps = DevOps + Security"

데브섹옵스는 데브옵스에 보안을 추가한 방법을 말한다. 최대한 빠르게 기능을 개발해 안정적으로 운영하는 데브옵스에 규제 문제와 관련된 보안을 적용하는 방법으로 데브옵스에서 한 단계 더 나아간 개발 방법이다.

---
2) 네트워크로 제공되는 앱・웹 서비스를 실제 운용 환경에서 이용할 수 있는 상태로 만드는 것
3) 조직 구성원 모두가 제반 법규를 철저하게 준수하도록 사전적, 상시적으로 통제, 감독하는 체제

4. 출처

https://blog.naver.com/azure0777/221066646741
https://brainbackdoor.tistory.com/53
http://blog.remotty.com/blog/2014/01/28/lets-study-rest/
https://xyom.github.io/2017/12/31/Rest%20API/
http://12bme.tistory.com/25
http://bcho.tistory.com/914
http://bcho.tistory.com/914

http://guruble.com/%EB%A7%88%EC%9D%B4%ED%81%AC%EB%A1%9C%EC%84%9C%EB%B9%84%EC%8A%A4microservice-%EC%95%84%ED%82%A4%ED%85%8D%EC%B2%98-%EA%B7%B8%EA%B2%83%EC%9D%B4-%EB%AD%A3%EC%9D%B4-%EC%A4%91%ED%97%8C%EB%94%94/
https://terms.naver.com/entry.nhn?docId=3548871&cid=42346&categoryId=42346
http://bcho.tistory.com/948

https://terms.naver.com/entry.nhn?docId=2842709&cid=40942&categoryId=32837
https://aws.amazon.com/ko/devops/what-is-devops/
https://ds5apn.wordpress.com/2014/12/28/backup-dev-kthcorp-com-devops-new-culture/
http://ithlim.blogspot.com/2017/11/sw-devops.html
http://techm.kr/bbs/board.php?bo_table=article&wr_id=1774
http://info104j56t5h.blogspot.com/2014/12/devops.html