

Rest를 알아보자

39ghwjd@naver.com

목차

1) URL, HTTP

2) Rest?

REST와 자주 거론되는 친구들!

- HTTP를 안 쓰는 REST 구조도 있습니다 -

URL, HTTP

1) URL, HTTP

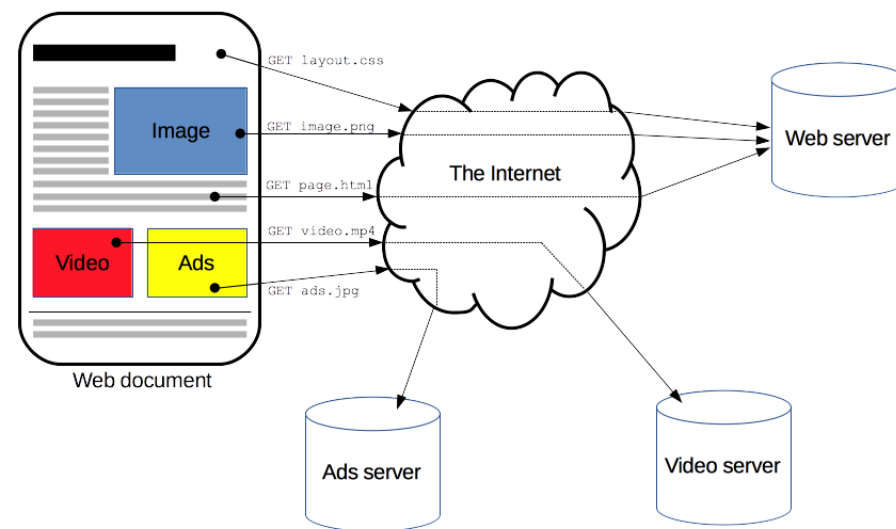
- URL -

```
https://developer.mozilla.org
https://developer.mozilla.org/en-US/docs/Learn/
https://developer.mozilla.org/en-US/search?q=URL
```

하나의 URL은 단 하나의 리소스만 식별

모든 리소스는 각각 자신의 URL을 가져야 한다
(주소 지정 가능성)

- HTTP -



- Statelessness : 무상태성 -

client, server가 각기 다른 상태를 저장
클라이언트가 어떤 상태인지
서버가 전혀 신경쓰지 않는다는 사실

1) URL, HTTP

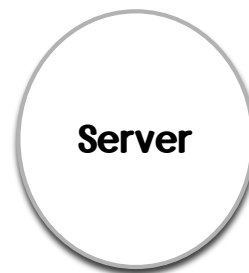
- Application 상태 -

클라이언트 측에서 관리,
서버는 가능한 상태 전이를 나
타내는 표현을 보내 조작 가능

Application State



Resource State



- 리소스 상태 -

서버에서 관리, 클라이언트
역시 서버에 원하는
새 상태를 설명하는 표현을
보내 조작 가능

- client 입장 -

리소스 상태에 직접적인 컨트롤을 전혀 할 수 없음,
모든 것은 다 서버 쪽에서 이루어짐

HTTP 세션 : 매우 짧음(하나의 요청동안)

서버는 클라이언트의

애플리케이션 상태를 전혀 알지 못함

1) URL, HTTP

HATEOAS(Hypermedia as the engine of application state)

서버가 다음에 무엇을 할 수 있는지
클라이언트에 설명하는 기법(하이퍼미디어)

+

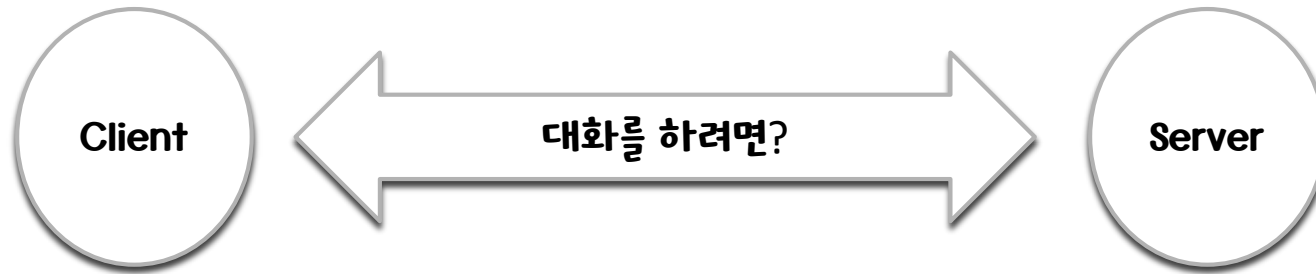
- 애플리케이션 상태의 엔진으로서 하이퍼미디어

애플리케이션 상태(어떤 페이지에 와 있는가?)

= 웹을 폼을 작성하고 링크를 따라가는 것으로 사용한다

1. 무엇이든 리소스가 될 수 있다

클라이언트와 서버가 무언가에 대한 대화를 하려면 그것을 부를 때 동의하는 이름



Resource? : 전자 문서, 데이터베이스의 행, 알고리즘 수행 결과 (컴퓨터에 저장할 수 있는 무언가)

유일한 제약 조건 - 모든 리소스는 URL을 가져야 한다는 것

1. 무엇이든 리소스가 될 수 있다

클라이언트 입장 : 리소스가 뭔지는 중요 X

클라이언트는 URL, 표현만 볼 뿐 직접 리소스를 보는 경우는 없기 때문

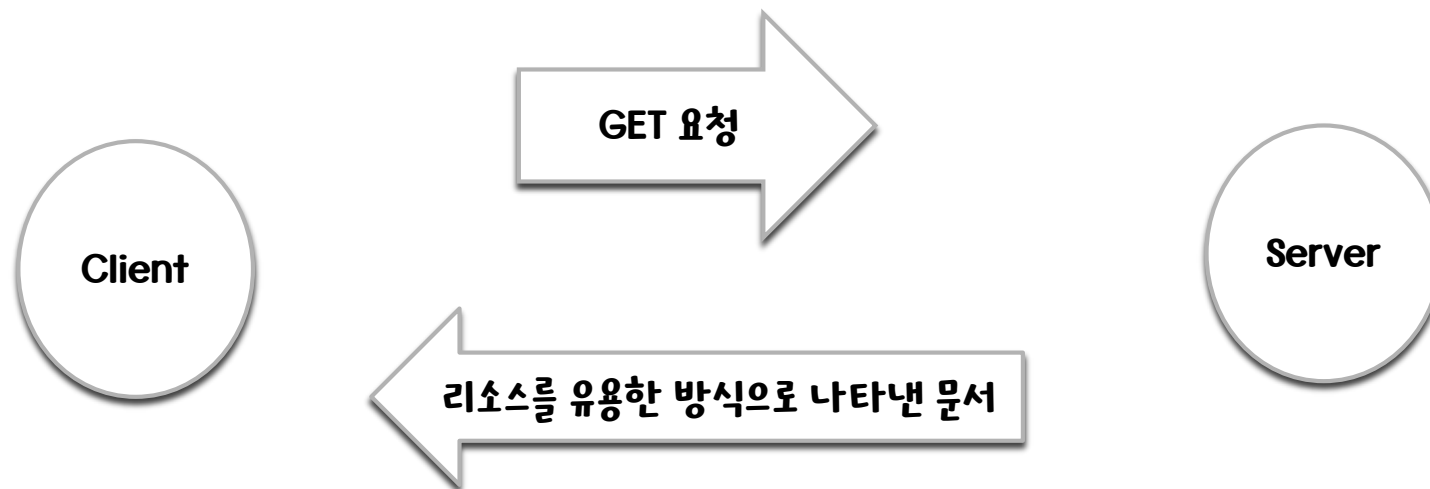
URL을 사용해 각 리소스에 전 세계에서 고유한 주소를 부여함
(무언가에 URL을 부여하면 그게 리소스가 됨)

2. 표현은 리소스 상태를 설명한다

- 표현(Representation) -

리소스의 현재 상태를 기계가 읽을 수 있는 설명으로 나타낸 것

표현은 리소스에 대한 어떤 정보든 담을 수 있고,
기계가 읽을 수 있는 어떠한 문서이든 상관없음



3. 표현은 양방향으로 전송된다

새 리소스가 어떻게 보여야 하는지에 대한 client의 생각을 보여줌

서버의 역할은 이 리소스를 생성 or 생성을 거절하는 것

클라이언트의 표현은 그저 제안, 서버는 그 표현을 추가 or 변경 or 일부를 무시할 수 있다



3. 표현은 양방향으로 전송된다

GET 요청 - 표현을 요청하고 있는 것

POST, PUT, PATCH 요청 - 클라이언트가 서버에 표현을 보냄
서버는 받은 요청을 반영하도록 리소스 상태를 변경하는 작업을 수행



3. 표현은 양방향으로 전송된다

GET, HEAD, DELETE

vs

POST, PUT, PATCH

3. 표현은 양방향으로 전송된다

GET, HEAD, DELETE

vs

POST, PUT, PATCH

역등성

ex) $5 * 0 = 0$

3. 표현은 양방향으로 전송된다 - 표현의 상태 전송

서버 : 리소스의 상태를 나타내는 표현을 보냄

클라이언트 : 그 리소스가 가졌으면 좋은 상태를 설명하는 표현을 보냄

4. 많은 표현이 있는 리소스

정부 문서,
특정 API
등

JSON, XML, 개요형 표현, 상세 표현

하나의 리소스에 표현이 하나 이상 있을 수 있음

- 1. Content Negotiation -

클라이언트는 HTTP 헤더의 값을 기준으로

각 표현을 구분

한 개의 리소스도 여러 개의 URL로 식별될 수 있음

2. 그 리소스의 각 표현마다

다른 URL을 부여하는 것

서버는 이 URL 중 하나를 공식 or

표준 URL로 지정해야 함

REST - 건축가의 관점에서 살펴보자
하나씩 살펴보면 이해가 갈 것


REST

잠깐!

솔직히 제대로 이해하려고 하면 끝이 없습니다.

꾸준한 공부, 복습으로 준비합니다 ㄱ

REST



You!
설계자

1. 제약 없이 전체 시스템 요구사항으로 시작
2. 설계 공간을 차별화하고 시스템 동작에 영향을 미치는 힘이 자연스럽게 흐르도록 하기 위해
3. 시스템 요소에 제약 조건을 점진적으로 식별 & 적용

**건축 설계 프로세스의
공동적 관점 중 하나!**

REST

**효율적, 안정적이며 확장가능한 분산시스템을 가져올 수 있는 SW
Architecture 디자인 제약의 모음!**

REpresentational State Transfer

REST

1. Client-Server

2. Stateless

3. Cacheable

4. Uniform Interface

5. Layered System

6. Code on Demand (Optional)

REST - 1) Client <-> Server

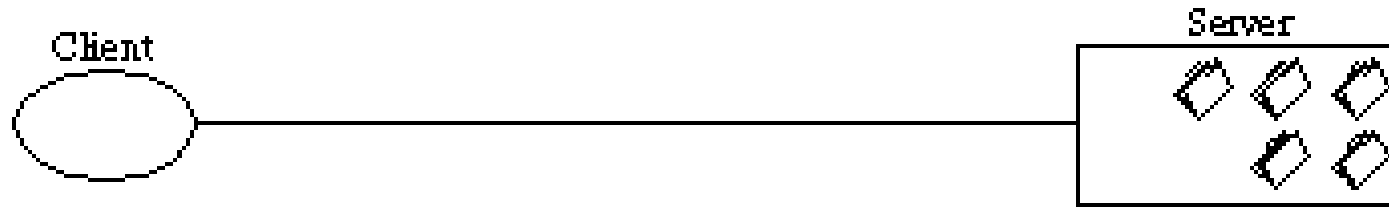


Figure 5-2. Client-Server

1.관심사의 분리

(사용자 인터페이스 문제 / 데이터 스토리지 문제)

분리를 통해 구성 요소를
독립적으로 발전시켜

2.이식성 개선 : 여러 플랫폼에서 사용자 인터페이스

여러 조직 도메인의

3.확장성 개선 : 서버 구성 요소를 단순화

인터넷 규모 요구 사항을 지원할 수 있다는 것

REST - 2) Stateless

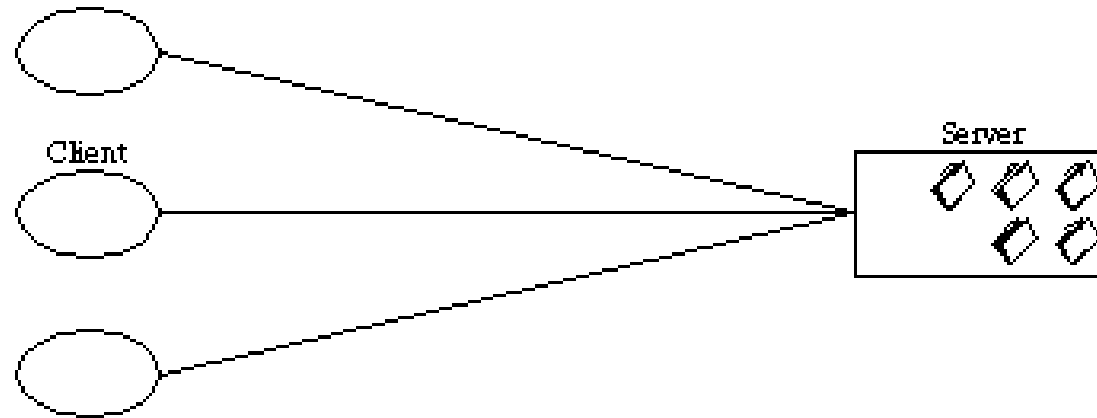


Figure 5-3. Client-Stateless-Server

Client - server에 대한 제약 조건 추가

**CSS(Client - stateless - server)와 같이 통신은
본질적으로 상태를 저장X**

client->server로 가는 각 요청은 이해할 수 있는

모든 정보들을 포함해야 함 &

서버에 저장된 conText를 이용할 수 없음

=> 세션 상태는 전적으로 Client에서 유지됨

REST - 2) Stateless

- 1. 가시성 -

모니터링 시스템이 요청의
전체 특성을 결정하기 위해

단일 요청 데이터 이상을 볼 필요가 X

- 2. 안정성 -

부분적 장애로부터
복구하는 작업을 용이하게 함

- 3. 확장성 -

요청 사이에
상태를 저장할 필요가 X

-> 서버 구성 요소가
리소스를 빠르게 해제 가능 &
서버가 요청 간 리소스 사용량을
관리할 필요가 없기 때문

=> 구현이 더욱 단순해짐

REST - 2) Stateless

- 단점 1 -

일련의 요청으로 전송되는
반복적인 데이터를 증가시켜

네트워크 성능을 저하시킬 수 있음
(데이터가 공유 컨텍스트에서
서버에 남아있을 수 없기 때문)

- 단점 2 -

Application State를
클라이언트 측에 배치하면
Application이 여러 client 버전에서
의미 체계의 올바른 구현에 종속

=> 일관된 Application 동작에 대한
서버의 제어 감소

REST - 3) Cache

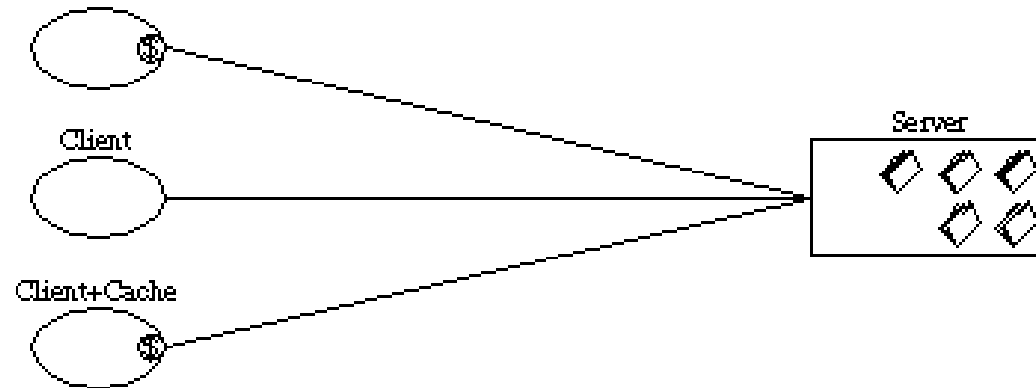


Figure 5-4. Client-Cache-Stateless-Server

네트워크 효율성 개선 목적

요청에 대한 응답 내의 데이터에
암시적 or 명시적으로 캐시 가능
or 불가능으로 label이 지정되어야 함

응답을 캐시할 수 있는 경우

클라이언트 캐시에는 해당 응답 데이터를
나중에 동일한 요청에 재사용할 수 있는
권한이 부여됨

REST - 3) Cache

- 단점 -

일부 상호작용을 부분 or

완전히 제거해 평균 대기 시간을 줄여

효율성, 확장성, 사용자가 인지하는

성능 향상 가능

캐시 내 오래된 데이터가

요청이 서버로 직접 전송되었을 때,
얻을 수 있었던 데이터와 크게 다를 경우

캐시가 안정성을 감소시킬 수 있음

REST - 4) Uniform Interface

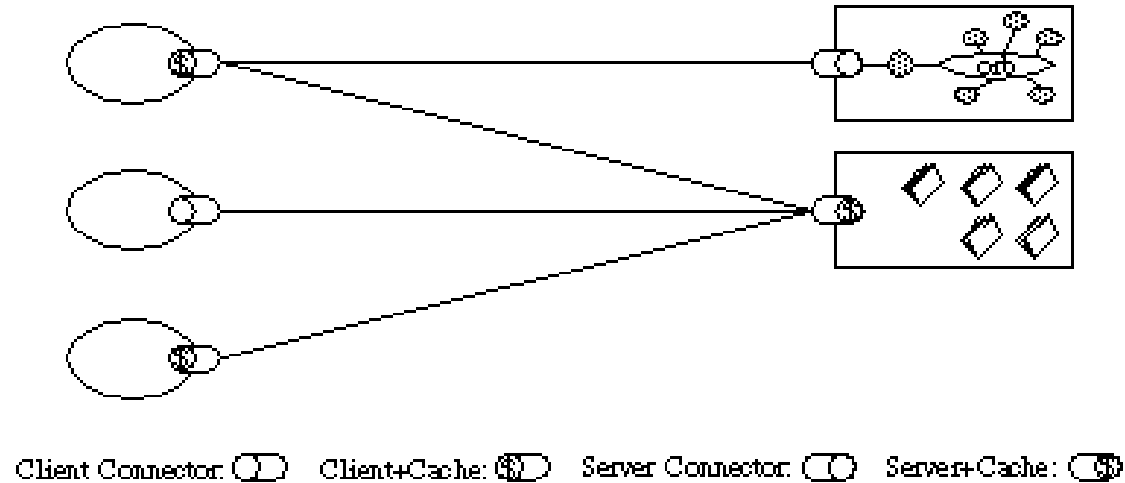


Figure S-6. Uniform-Client-Cache-Stateless-Server

일반적인 SW Engineering 원칙 + 구성 요소 인터페이스

=>전체 시스템 아키텍처의 단순화 &
상호 작용의 가시성 향상

구현과 제공하는 서비스를 분리 => 독립적인 발전 가능

단점 : 정보가 Application 요구에
특정한 형식이 아닌 표준화된 형식으로 전송
=> 균일한 인터페이스가 효율성을 저하시킴

REST - 5) Layered System

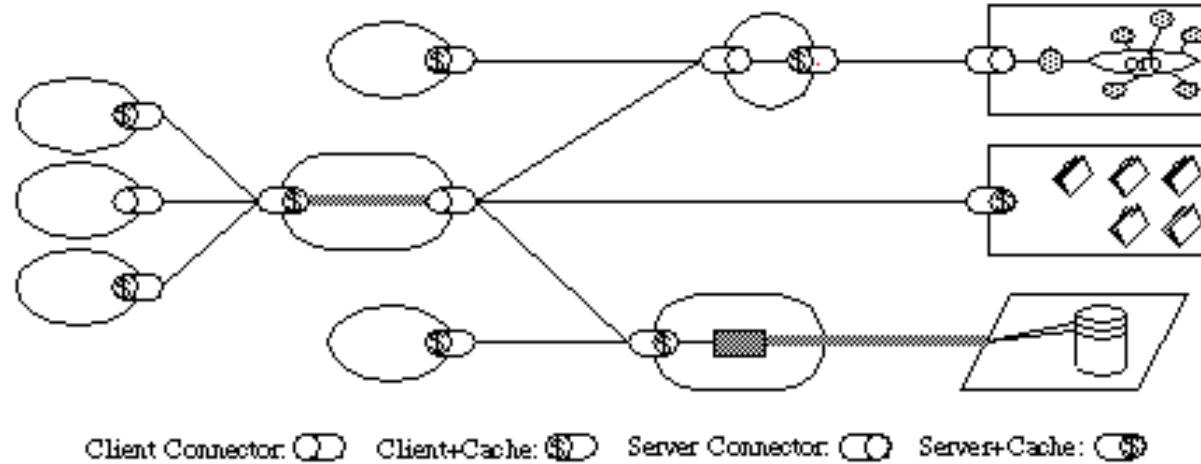


Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

인터넷 규모에 해당하는
요구 사항 동작 개선 목적

시스템 지식을 단일 계층으로 제한

각 구성 요소가 상호 작용하는 직접적인 계층
너머를 볼 수 없도록 구성 요소 동작을 제한

=> 전체 시스템 복잡성 제한 & 기판 독립성 촉진

REST - 5) Layered System

- Layer -

레거시 서비스를 캡슐화,
레거시 클라이언트로부터 새 서비스를 보호 &
자주 사용하지 않는 기능을 공유 중개자로 이동

=> 구성 요소를 단순화할 수 있음

- 중개자 -

여러 네트워크 및 프로세서에서
서비스의 로드 밸런싱을 가능하게 함

=> 시스템 확장성을 개선할 수 있음

REST - 5) Layered System

데이터 처리에 Overhead &
대기 시간을 추가해
사용자가 인식하는 성능을 감소시킨다는 것

- 해결책 -

공유 캐시를 배치하면
상당한 성능 이점 + 데이터에 보안 정책

REST - 6) 주문형 코드

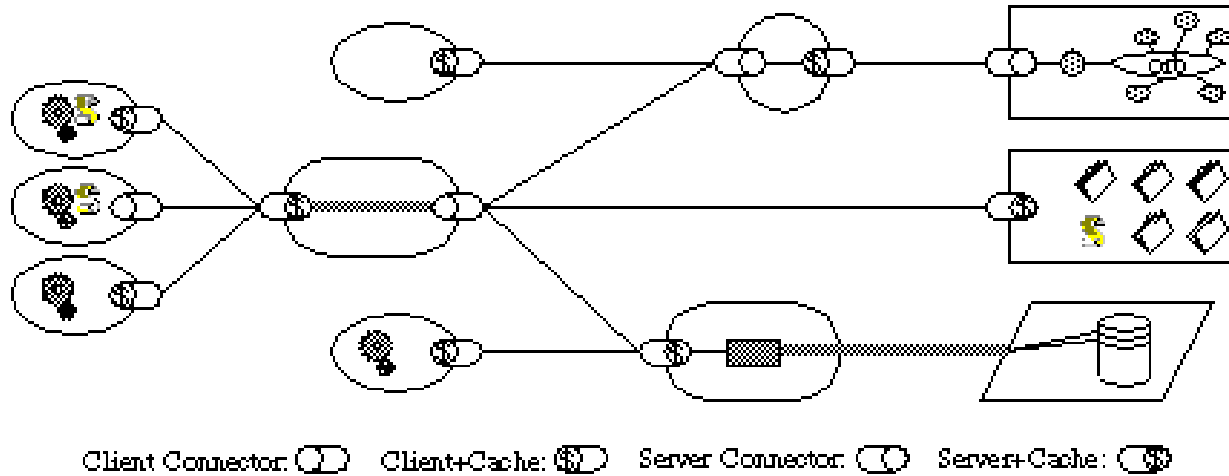


Figure 5-8. REST

applet 또는 script 형태의 코드를 다운, 실행
함으로써, 클라이언트 기능을 확장

사전 구현해야 하는 기능의 수를 줄여
클라이언트를 단순화함

배포 후 기능 다운로드를 허용
=> 시스템 확장성이 향상됨
- But, 가시성도 감소

Rest 내 선택적 제약 조건

참고

[MDN REST 설명](#)

[REST API Tutorial](#)

[REST ICS UCI EDU 문서](#)