Http/Https

HTTP(Hyper Text Transfer Protocol)란?

HTTP란 서버/클라이언트를 모델을 따라 데이터를 주고 받기 위한 프로토콜이다.

즉, HTTP는 인터넷에서 하이퍼텍스트를 교환하기 위한 통신 규약으로, 80번 포트를 사용하고 있다. 따라서 HTTP 서버가 80번 포트에서 요청을 기다리고 있으며 클라이언트는 80번 포트로 요청을 보내고 있다.

HTTP는 1989년 팀 버너스 리에 의해 처음 설계되었으며, WWW 기반에서 세계적인 정보를 공유하는 데 큰 역할을 하였다.

HTTP의 역사

HTTP/0.9 - 1991년 : GET 메소드만 지원, HTTP 헤더 X

HTTP/1.0 - 1996년 : 메소드, 헤더 추가

HTTP/1.1 - 1997년 : 현재 가장 많이 사용하는 버전

HTTP/2 - 2015년 : 성능 개선

HTTP/3 - 현재 진행중 : TCP 대신 UDP 사용, 성능 개선

HTTP의 특징

HTTP는 다음과 같은 특징들을 가지고 있다.

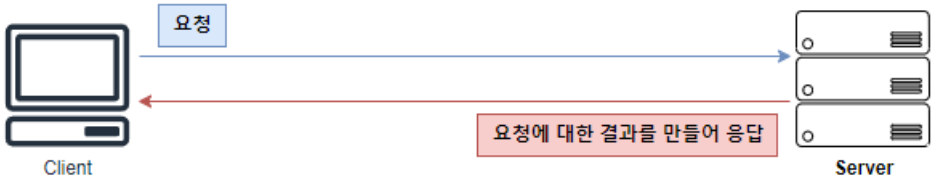
- **클라이언트 - 서버 구조 (Request - Response 구조)**
- **무상태성(Stateless)**
- **비연결성(Connectionless)**
  - 지속 연결

클라이언트 - 서버 구조란?

Request와 Response 구조로,

클라이언트는 HTTP 메세지를 통해 서버에 요청을 보내고, 서버에서 응답이 올 때까지 기다린다.

서버는 요청에 대한 결과를 만들어서 클라이언트에게 응답한다.



"클라이언트 - 서버" 구조를 만드는 이유는 클라이언트는 서버 내부의 구조를 알지 못하기 때문에

서버의 구조가 바뀐다고 해서 클라이언트에게 영향이 가지 않게 해야 하기 때문이다.

이렇게 클라이언트 서버 구조를 만들게 되면, 서버의 구조가 바뀌더라도 클라이언트에게 아무런 영향이 가지 않아 **양쪽이 독립적으로 진행**할 수 있게 된다.

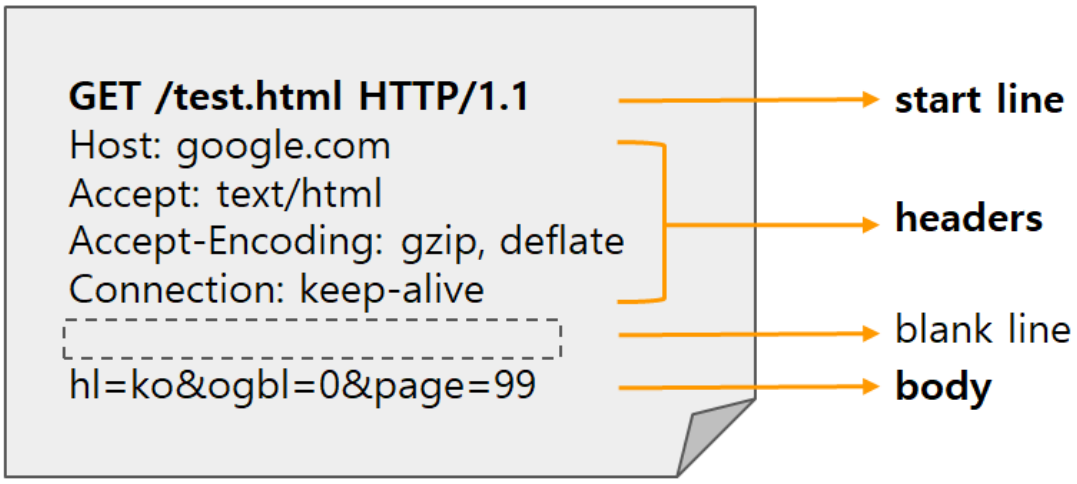
Request Message

HTTP Request Message는 공백(blank line)을 제외하고 3가지 부분으로 나뉘어진다.

HTTP Request Message의 구조

- Start Line

- Headers
- Body



Start Line

HTTP Request Message의 시작 라인

HTTP Request의 start line 3가지 부분으로 구성

- HTTP method
- Request target
- HTTP version

```
GET /test.html HTTP/1.1
[HTTP Method] [Request target] [HTTP version]
```

[HTTP method]는 요청의 의도를 담고 있는 GET, POST, PUT, DELETE 등이 있다.

- GET은 존재하는 자원에 대한 요청
- POST는 새로운 자원을 생성
- PUT은 존재하는 자원에 대한 변경
- DELETE는 존재하는 자원에 대한 삭제와 같은 기능을 가지고 있다.

[Request target]은 HTTP Request가 전송되는 목표 주소이다.

[HTTP version]은 version에 따라 Request 메세지 구조나 데이터가 다를 수 있어서 version을 명시한다.

Header

해당 Request에 대한 추가 정보를 담고 있는 부분

예를 들어, Request 메세지 body의 총 길이 등 key : value 형태로 구성

headers도 크게 3가지 부분으로 나뉨(General headers, request headers, entity headers)

```
Host: google.com
Accept: text/html
Accept-Encoding: gzip, deflate
Connection: keep-alive
...
```

[Host] 요청하려는 서버 호스트 이름과 포트번호

[User-agent] 클라이언트 프로그램 정보, 이 정보를 통해 서버는 클라이언트 프로그램(브라우저)에 맞는 최적의 데이터를 보내줄 수 있다.

[Referer] 바로 직전에 머물렀던 웹 링크 주소

[Accept] 클라이언트가 처리 가능한 미디어 타입 종류 나열

[If-Modified-Since] 여기에 쓰여진 시간 이후로 변경된 리소스 취득. 페이지가 수정되었으면 최신 페이지로 교체한다.

[Authorization] 인증 토큰을 서버로 보낼 때 쓰이는 Header

[Origin] 서버로 Post 요청을 보낼 때 요청이 어느 주소에 시작되었는지 나타내는 값. 이 값으로 요청을 보낸 주소와 받는 주소가 다르면 CORS에러가 발생한다.

[Cookie] 쿠키 값이 key-value로 표현된다.

Body

HTTP Request가 전송하는 데이터를 담고 있는 부분

전송하는 데이터가 없다면 body 부분은 비어있습니다.

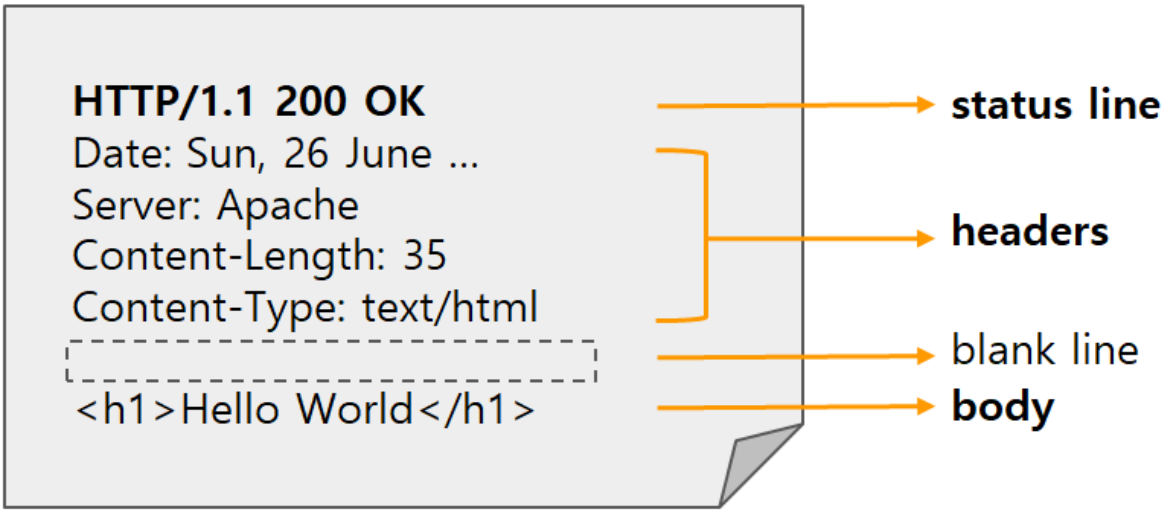
보통 post 요청일 경우, HTML 폼 데이터가 포함되어 있다.

```
POST /test HTTP/1.1

Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 83
Content-Type: application/json
Host: google.com
User-Agent: HTTPie/0.9.3

{
    "test_id": "tmp_1234567",
    "order_id": "8237352"
}
```

Response Message

HTTP Response Message는 request와 동일하게 공백(blank line)을 제외하고 3가지 부분으로 나누어진다.

- Status Line
- Headers
- Body



Status Line

HTTP Response의 상태를 간략하게 나타내주는 부분이다.

HTTP Response의 status line 또한 3가지 부분으로 구성되어 있다.

- HTTP version
- Status Code
- Status Tex

```
HTTP/1.1 200 OK
[HTTP version] [Status Code] [Status Text]
```

Headers

Request의 headers와 동일하다.

다만 Response에서만 사용되는 header 값들이 있다.

예를 들어, User-Agent 대신에 Server 헤더가 사용된다.

Body

Response의 body와 일반적으로 동일하다.

Request와 마찬가지로 모든 Response가 body를 가지고 있지는 않다.

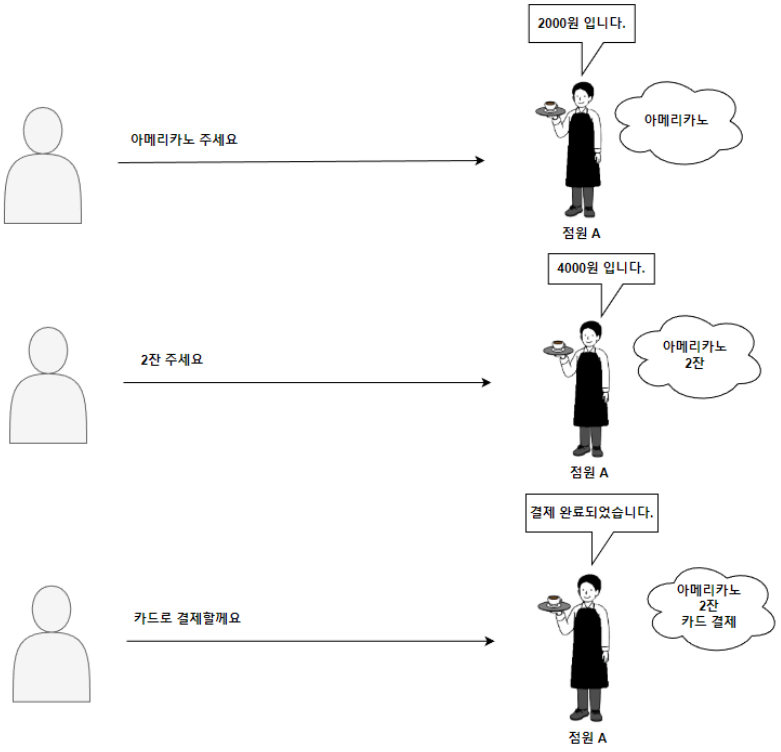
데이터를 전송할 필요가 없을 경우 body가 비어있게 된다.

Stateless(무상태성)과 Stateful(상태성)

상태성은 말 그대로 상태를 유지하는 것이다.
상태를 유지하기 때문에 클라이언트와 연결된 서버는 클라이언트 요청에 대한 정보를 계속 갖고 있게 된다. 하지만, 이 연결된 서버가 장애가 발생해 작동하지 않는다면 클라이언트는 다른 서버와 다시 연결을 하고 다시 요청을 해야 한다.

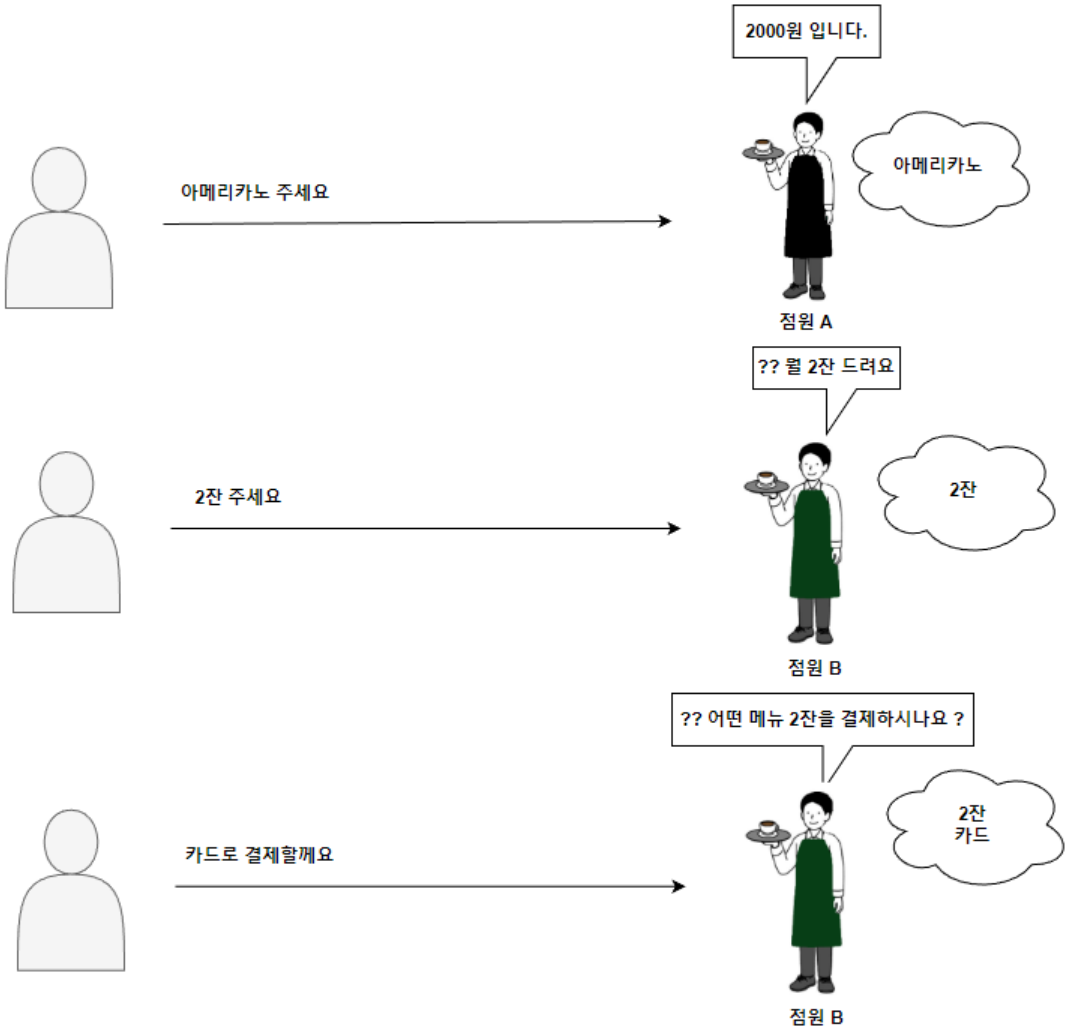
- 카페에서 같은 점원에게 커피를 주문할 경우의 예를 들어 보자.

이 경우 **점원 A는 고객의 주문 상태에 대해 기억**하고 있다.



- 하지만 중간에 점원이 바뀔 경우의 예를 들어 보자.

점원 A에서 점원 B로 바뀌면서 점원 A가 고객의 주문을 B에게 알려주지 않았기 때문에 **점원 B는 이전의 고객 주문 정보를 알 수 없어 아래와 같은 상황이 발생**한다.



이러한 상황이 발생할 수 있는 문제가 있기 때문에 **Stateful은 확장성에 한계**를 갖는다.

Stateful은 보통 상태 유지가 필요한 "**로그인 기능**"에 **브라우저 쿠키**와 **서버 세션** 등을 사용해 **상태를 유지할 수 있도록 설계**한다.

Stateless(무상태성)

말 그대로 상태를 유지하지 않는 것이다.

상태를 유지하지 않기 때문에 클라이언트와 연결된 서버가 장애가 발생했더라도 클라이언트가 요청 정보를 기억하기 때문에 다른 서버를 연결하여 문제가 생기지 않는다.

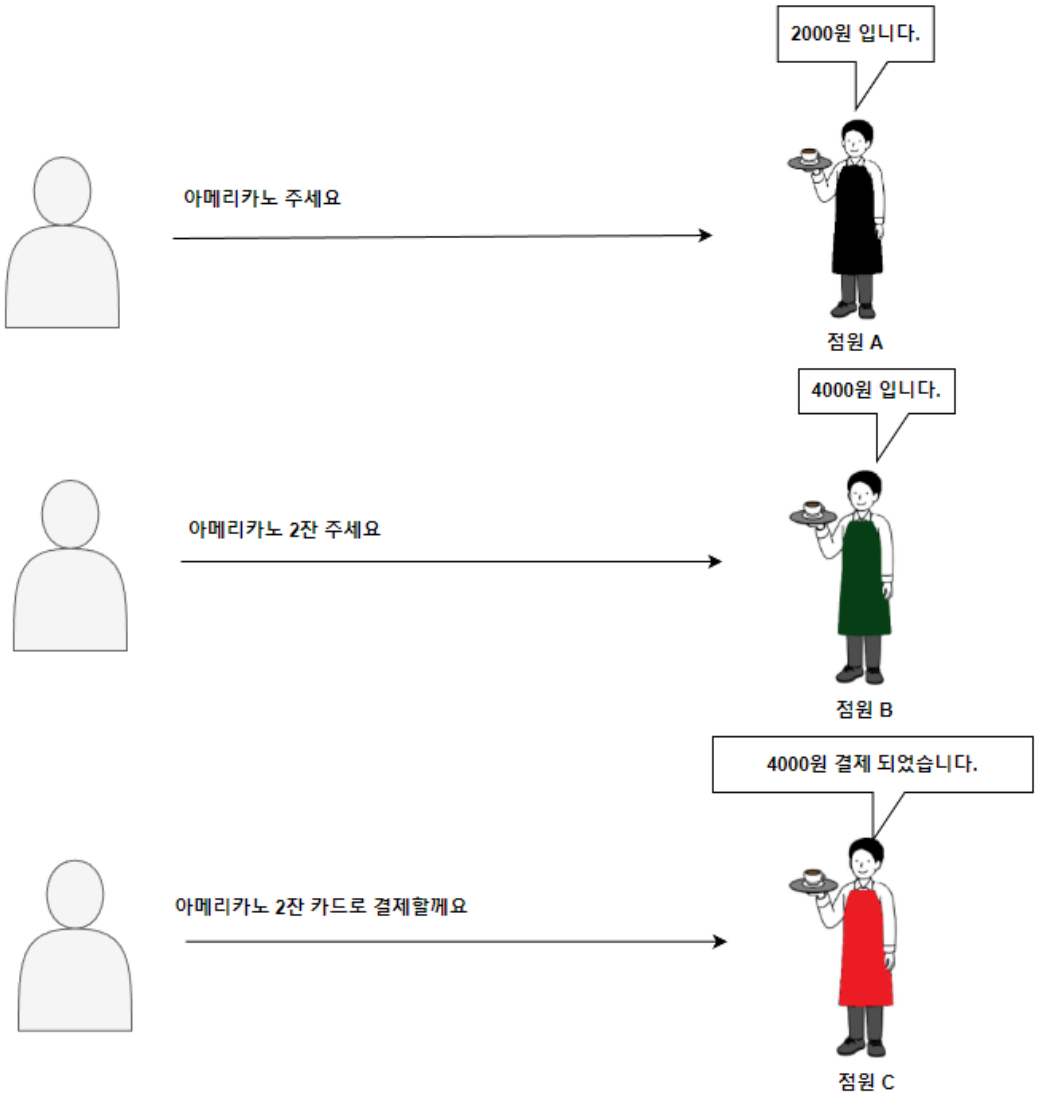
이러한 이유로 Stateless는 수평확장에 유리하다.

하지만 클라이언트가 Stateful에 비해 요청을 한번에 보내기 때문에 **데이터를 너무 많이 보낸다는 단점**이 있다.

- 위와 똑같은 예시로 카페에서 점원에게 커피를 주문할 경우를 예를 들어 보자.

Stateless는 고객이 주문 정보를 기억하고 있기 때문에 중간에 다른 점원으로 바뀌어도 주문을 할 수 있다.

만약, 고객의 요청이 증가하더라도 여러 점원을 투입할 수 있기 때문에 확장성이 용이하다.



Stateless에서 상태 유지가 필요한 경우라면 어떻게 해야할까?

Connections(연결성)과 Connectionless(비연결성)

TCP / IP의 경우에는 기본적으로 연결을 유지하는 특징을 가지고 있다.

연결을 유지하는 경우에는 클라이언트와 서버가 한 번 연결이 되었다면 클라이언트가 요청을 보내지 않더라도 계속 연결을 유지해야 한다.

그렇기 때문에 **연결을 유지하는 서버의 자원이 계속 소모**되게 된다.

Connectionless(비연결성)

HTTP는 기복적으로 연결을 유지하지 않는 모델로 일반적으로 **초 단위 이하의 빠른 속도로 응답**한다.

동작 방식은 클라이언트가 서버에 요청을 보내고 서버가 응답을 주었다면 TCP / IP 연결을 끊어 **최소한의 자원을 사용해 서버를 유지**할 수 있다. (서버 자원을 효율적으로 사용)

HTTP 초기에는 각각의 자원들을 다운로드하기 위해 연결과 종료를 반복해야 하는 문제가 있었지만, HTTP 지속 연결로 연결이 이루어지고 난 뒤 각각의 자원들을 요청하고 모든 자원에 대한 응답을 받은 후에 연결을 종료하여 초기 문제가 해결되었다.

HTTP/2와 HTTP/3에는 **HTTP 지속 연결**이 더욱 최적화 되어있다.

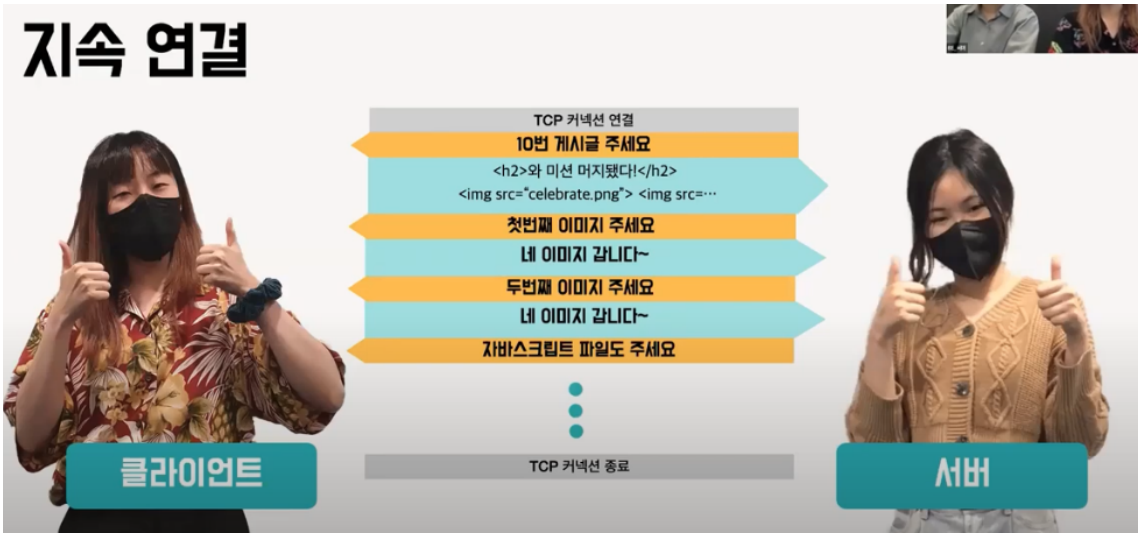
지속 연결

초기 HTTP는 비지속 연결을 사용했다.

요청을 주고 받을 때마다 새로운 TCP Connection을 연결하고 종료하는 것이다.

이는 데이터가 작을 때는 문제가 없었지만 주고받는 데이터가 많아지면서 연결에 사용되는 자원의 낭비와 속도 저하 등의 문제가 생겼다.

그래서 지속 연결을 통해서 서버의 부하를 줄이고 통신을 좀 더 빠르게 할 수 있게 되었다.



여기서 더 나아가서 Response 응답을 기다리지 않고 바로 다음 리퀘스트를 보내는 **파이프라이닝**도 가능해졌다.

하지만 **파이프라이닝**은 성능 향상이 미미하고 프록시 버그가 많아서 모던 브라우저에서는 기본적으로 활성화 되어 있지 않다.

HTTP 메서드

멱등성 : 수학에서 연산을 여러번 적용해도 경과가 달라지지 않는 성질을 의미한다.

HTTP에서는 여러번 요청한 결과 서버의 상태가 항상 동일한 때를 의미한다.

메서드 별로 역등성 성립 여부가 다르다!!

GET

데이터를 가져올 때만 사용해야 한다.

본문을 담는 것이 금지되어 있지는 않지만 구현된 서비스에서 요청을 거부할 수도 있기 때문에 담지 않는 것이 좋다.

GET은 여러 번 실행해도 특정 리소스를 보내주기만 하는 것이라 당연히 멱등성이 보장된다.

POST

대상 리소스에게 Request 본문을 해당 리소스의 Semantic에 따라 처리하도록 요청하는 메서드이다.

게시판, 블로그 같은 글 모음에 글 보내기, 서버에 새로운 리소스 생성하기 등에 사용할 수 있다.

게시글을 계속 보내면 글이 쌓이게 되고, 이는 서버의 상태의 변화를 의미하므로 멱등하지 않다.

PUT / PATCH

PUT은 대상 리소스가 없다면 생성하고 있다면 Request의 본문대로 교체하는데 사용된다.

PATCH는 리소스의 일부를 수정하는데에 사용된다.

PUT은 멱등하지만 PATCH는 멱등할 수도 있고 아닐 수도 있다.

- PATCH는 rfc2616 스팩 상으로는 멱등성을 보장하지 않았다. 하지만 복수의 PATCH 요청이 부작용을 일으키지 않도록 막는 목적으로 멱등하게 처리해도 된다고 rfc5789에 추가 기술되어 있다.

DELETE

DELETE도 멱등성을 보장해야 한다.

그러므로 게시판의 마지막 글 제거를 하기 위해 DELETE를 사용하면 안된다. 왜냐하면 실행 결과가 계속 달라지기 때문이다.

그 외에도 GET으로 요청했을 때의 응답 헤더만 가져오는 HEAD,

해당 리소스의 통신 옵션만 가르쳐주는 OPTIONS 등이 있다.

HTTPS(Hyper Text Transfer Protocol Secure)란?

하지만 HTTP는 암호화가 되지 않은 평문 데이터를 전송하는 프로토콜이였기 때문에, HTTP로 비밀번호나 주민등록번호 등을 주고 받으면 제 3자가 정보를 조회할 수 있었다. 이러한 문제를 해결하기 위해 HTTPS가 등장하게 되었다.

HTTPS란?

Hyper Text Transfer Protocol over Secure Socket Layer, HTTP over TLS, HTTP over SSL, HTTP Secure 등으로 불리는 HTTPS는 **HTTP에 데이터 암호화가 추가된 프로토콜**이다. HTTP와 다르게 443 포트를 사용하며, 네트워크 상에서 중간에 제 3자가 정보를 볼 수 없도록 암호화를 지원하고 있다.

대칭키 암호화와 비대칭키 암호화

HTTPS는 대칭키 암호화 방식과 비대칭키 암호화 방식을 모두 사용하고 있다. 각각의 암호화 방식은 다음과 같다.

- 대칭키 암호화
  - 클라이언트와 서버가 동일한 키를 사용해 암호화/복호화를 진행함
  - 키가 노출되면 매우 **위험하지만 연산 속도가 빠름**
- 비대칭으로 암호화
  - 1개의 쌍으로 구성된 공개키와 개인키를 암호화/복호화 하는데 사용함

- 키가 노출되어도 비교적 **안전하지만 연산 속도가 느림**

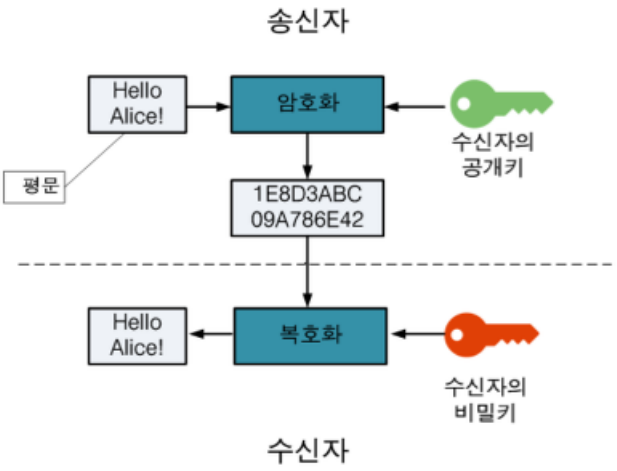
대칭키는 비교적 쉬운 개념이므로, 비대칭키 암호화에 대해 조금 자세히 살펴보도록 하자.

비대칭키 암호화는 공개키/개인키 암호화 방식을 이용해 데이터를 암호화하고 있다. 공개키와 개인키는 서로를 위한 1쌍이다.

- 공개키: 모두에게 공개가능한 키
- 개인키: 나만 가지고 알고 있어야 하는 키

암호화를 공개키로 하느냐 개인키로 하느냐에 따라 얻는 효과가 다른데, 공개키와 개인키로 암호화하면 각각 다음과 같은 효과를 얻을 수 있다.

- 공개키 암호화 : 공개키로 암호화를 하면 개인키로만 복호화(decoding)할 수 있다. → 개인 키는 나만 가지고 있으므로, **나만 볼 수 있다**.
- 개인키 암호화 : 개인키로 암호화하면 공개키로만 복호화할 수 있다. → 공개키는 모두에게 공개되어 있으므로, 내가 인증한 정보임을 알려 **신뢰성을 보장**할 수 있다.



HTTPS의 동작 과정

HTTPS는 대칭키 암호화와 비대칭키 암호화를 모두 사용하여 빠른 연산 속도와 안전성을 모두 얻고 있다.

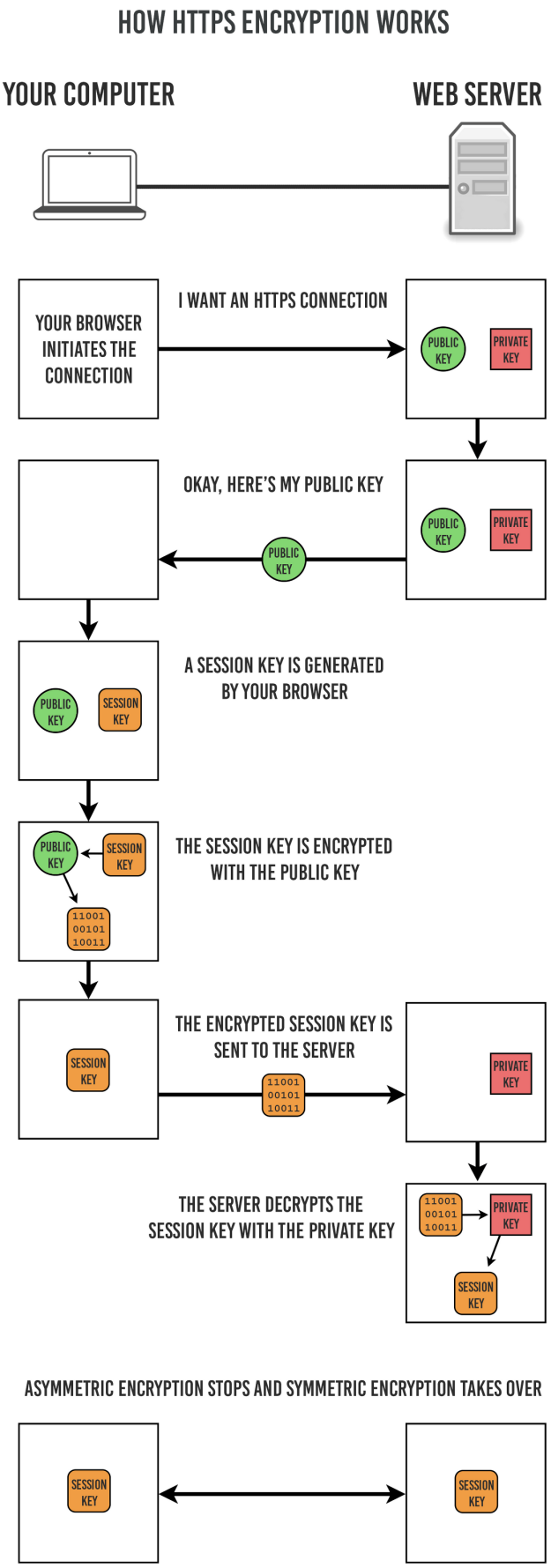
HTTPS 연결 과정(hand-shaking)에서는 먼저 서버와 클라이언트 간에 세션키를 교환한다. 여기서 세션키는 주고받는 데이터를 암호화하기 위해 사용되는 대칭키이며, 데이터 간의 교환에는 빠른 연산 속도가 필요하므로 세션키는 대칭키로 만들어진다. 문제는 이 세션키를 클라이언트와 서버가 어떻게 교환할 것이냐인데, 이 과정에서 비대칭키가 사용된다.

즉, 처음 연결을 성립하여 안전하게 세션키를 공유하는 과정에서 비대칭키가 사용되는 것이고, 이후에 데이터를 교환하는 과정에서 빠른 연산 속도를 위해 대칭키가 사용되는 것이다.

실제 HTTPS 연결 과정이 성립되는 흐름을 살펴보면 다음과 같다.

1. 클라이언트(브라우저)가 서버로 최초 연결 시도를 함
2. 서버는 공개키(엄밀히는 인증서)를 브라우저에게 넘겨줌
3. 브라우저는 인증서의 유효성을 검사하고 세션키를 발급함
4. 브라우저는 세션키를 보관하며 추가로 서버의 공개키로 세션키를 암호화하여 서버로 전송함
5. 서버는 개인키로 암호화된 세션키를 복호화하여 세션키를 얻음

클라이언트와 서버는 동일한 세션키를 공유하므로 데이터를 전달할 때 세션키로 암호화/복호화를 진행함


HOW HTTPS ENCRYPTION WORKS
YOUR COMPUTER
WEB SERVER
YOUR BROWSER INITIATES THE CONNECTION
I WANT AN HTTPS CONNECTION
PUBLIC KEY
PRIVATE KEY
OKAY, HERE'S MY PUBLIC KEY
A SESSION KEY IS GENERATED BY YOUR BROWSER
SESSION KEY
THE SESSION KEY IS ENCRYPTED WITH THE PUBLIC KEY
11001 00101 10011
THE ENCRYPTED SESSION KEY IS SENT TO THE SERVER
THE SERVER DECRYPTS THE SESSION KEY WITH THE PRIVATE KEY
ASYMMETRIC ENCRYPTION STOPS AND SYMMETRIC ENCRYPTION TAKES OVER

HTTPS의 발급 과정

위의 과정에서 추가로 살펴봐야 할 부분은 서버가 비대칭키를 발급받는 과정이다. 서버는 클라이언트와 세션키를 공유하기 위한 공개키를 생성해야 하는데, 일반적으로는 인증된 기관에 공개키를 전송하여 인증서를 발급받는다. 자세한 과정은 다음과 같다.

1. A기업은 HTTP 기반의 어플리케이션에 HTTPS를 적용하기 위해 공개키/개인키를 발급함
2. CA 기업에게 돈을 지불하고, 공개키를 저장하는 인증서의 발급을 요청함.
3. CA 기업은 서버(A 기업)의 공개키, 서버(A 기업)의 정보 등을 기반으로 인증서를 생성하고, CA 기업의 개인키로 암호화하여 A 기업에게 이를 제공함
4. A 기업은 클라이언트에게 암호화된 인증서를 제공함
5. 브라우저는 CA 기업의 공개키를 미리 다운받아 갖고 있어, 암호화된 인증서를 복호화함
6. 암호화된 인증서를 복호화하여 얻은 A기업의 공개키로 세션키를 공유함

인증서는 CA의 개인키로 암호화되었기 때문에, 신뢰성을 확보할 수 있고, 클라이언트는 A기업의 공개키로 데이터를 암호화하였기 때문에 A기업만 복호화하여 원본의 데이터를 얻을 수 있다. 여기서 인증서에는 A 기업의 공개키가 포함되어 있으므로, A 기업의 공개키라고 봐도 무방하다. 또한 브라우저에는 인증된 CA기관의 정보들이 사전에 등록되어 있어 인증된 CA 기관의 인증서가 아닌 경우에는 다음과 같은 형태로 브라우저에서 보여지게 된다.

HTTP와 HTTPS

HTTP는 암호화가 추가되지 않았기 때문에 보안에 취약한 반면, HTTPS는 안전하게 데이터를 주고받을 수 있다. 하지만 HTTPS를 이용하면 암호화/복호화의 과정이 필요하기 때문에 HTTP보다 속도가 느리다. (물론 오늘날에는 거의 차이를 못 느낄 정도이다.) 또한 HTTPS는 인증서를 발급하고 유지하기 위한 추가 비용이 발생한다.

그렇다면 언제 HTTP를 쓰고, 언제 HTTPS를 쓰는 것이 좋겠는가?

개인 정보와 같은 민감한 데이터를 주고 받아야 한다면 HTTPS를 이용해야 하지만, 노출이 되어도 괜찮은 단순한 정보 조회만을 처리하고 있다면 HTTP를 이용하면 된다.

References :

https://mangkyu.tistory.com/98

https://hahahoho5915.tistory.com/62

https://hstory0208.tistory.com/entry/HTTP란-무상태성과-비연결성에-대해-알아보자

https://www.youtube.com/watch?v=IjxkKQvn8Bc