



HTTP, HTTPS

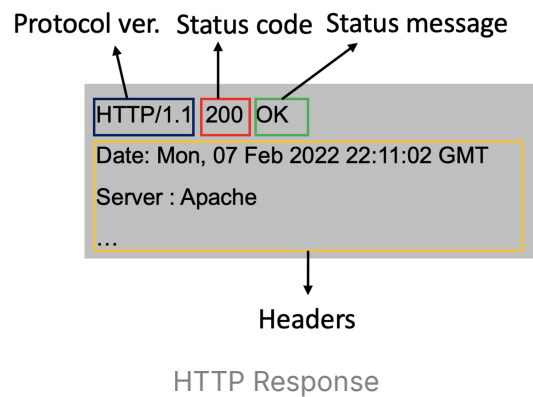
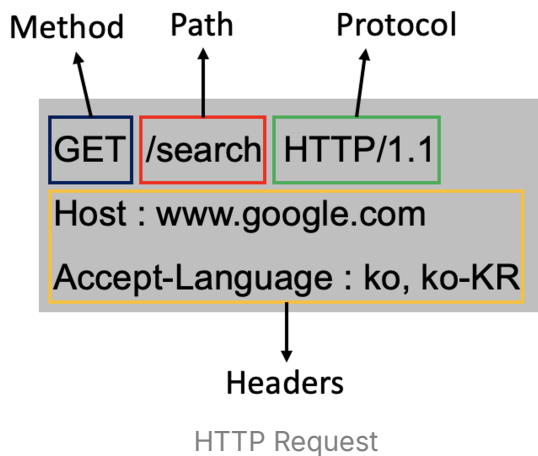


HTTP (HyperText Transfer Protocol)



서버-클라이언트 모델을 따르면서 request/response 구조로 웹(www) 상에서 정보를 주고받는 프로토콜

- HTML과 같은 하이퍼텍스트 문서를 네트워크 상에서 주고 받기 위한 통신 규약
- TCP/IP 기반으로 작동
- 클라이언트가 서버에 HTTP request를 보내면 서버가 클라이언트에게 HTTP response를 보내는 구조
- request message
 - start line (method, path, HTTP version)
 - headers
 - body
- response message
 - status line(HTTP version, status code, status message)
 - headers
 - body





- HTTP Version
 - HTTP/1.1: 가장 많이 사용, 대부분의 기능을 포함 (TCP)
 - HTTP/2: 성능 개선 (TCP)
 - HTTP/3: 현재 진행형, 성능 개선 (UDP)

HTTP의 특징

Connectionless (비연결성)

- 클라이언트가 서버에 요청을 하고 응답을 받으면 바로 TCP/IP 연결을 끊어버리는 것

 기본적으로 TCP/IP는 연결을 종료하지 않고 유지함 → 자원 낭비 발생 가능성 

- 서버의 자원을 효율적으로 관리
- 동시 접속을 최소화하여 더 많은 요청 처리 가능
- 발생할 수 있는 문제
 - 클라이언트의 이전 상태(로그인 유무 등)를 알 수 없는 **stateless** 특성 발생
 - 연결이 끊어짐에 따라 새로 연결될 때 TCP/IP 연결을 새로 맺어야 하므로 **3-way handshake**에 따른 시간 추가
 - HTML 뿐만 아니라 JavaScript, CSS, image 등 수 많은 자원이 함께 다운로드 됨

- 해결
 - 현재는 HTTP **지속 연결(Persistence Connections)** 로 문제 해결

지속 연결(= HTTP keep-alive, HTTP connection reuse) : 요청에 따라 연결된 이후 일정 시간 연결을 유지하거나 여러 개의 요청 (HTML, 자바스크립트, 이미지 등)에 대한 응답이 다 올때까지 기다린 후 연결을 종료하는 것

- HTTP/1.0 부터 적용. HTTP/2, HTTP/3에서 최적화를 이룸

Stateless (무상태)

- 통신이 끝나면 서버는 더 이상 클라이언트의 이전 상태를 유지하지 않음
- 발생할 수 있는 문제
 - 사이트에 로그인 시, 이전의 로그인이 유지되지 않아 매번 로그인을 해야함
 - 팝업 보지 않기를 체크했는데, 페이지를 유지할 때마다 매번 팝업창이 뜸
- 상태를 유지하는 **stateful**의 경우 항상 같은 서버가 유지되어야 하지만, **stateless** 는 클라이언트의 요청에 어떤 서버가 응답해도 상관 없다. ➡ 클라이언트의 요청이 대폭 증가해도 서버 증설로 해결 가능



Connectionless, Stateless 특성을 가진 HTTP의 단점을 해결하기 위해 cookie, session, jwt 등을 도입

!? 왜 HTTP는 Stateless 구조를 채택할까?

- 단순성, 확장성, 무상태 연결, 성능, 데이터 일관성 때문
- 각 요청은 서로 독립적. 서버가 요청에 대한 정보를 기억할 필요가 없으므로 개발과 유지보수가 편리
- 서버가 각 클라이언트의 상태를 추적할 필요가 없으므로 대규모 서비스에 적합. 서버 자원을 효율적으로 사용할 수 있고, 로드 밸런싱을 쉽게 구현 가능

- 각 요청과 응답이 독립적이므로 여러 서버가 동일한 요청 처리 가능. 분산 시스템에서 핵심
- 서버가 빠르게 응답 가능
- 여러 서버 사이에서 데이터 일관성 문제가 발생할 가능성 감소
- 초기 HTTP/1.0에서는 각 요청 후 연결을 종료하는 것이 일반적이었으나, 이러한 접근 방식은 많은 TCP 연결 및 종료로 인한 오버헤드와 성능 저하를 초래함. 이에 따라 HTTP/1.1에서는 TCP 연결을 유지하는 Keep-Alive 기능이 기본값으로 설정되어, 연결 설정에 대한 오버헤드를 줄이고 전체적인 통신 효율성을 향상.

HTTP request method

GET

```
GET /user/1
```

- 클라이언트가 서버에 리소스를 요청
- 필요한 정보를 특정하기 위해 URL 주소 끝에 `Query String(=Query Parameter)` 을 추가하여 정보를 조회

`Query String` : URL 주소 끝에 key-value 쌍으로 parameter를 포함하여 전송하는 부분

ex) `https://www.google.com/search?q=get`

- URL에 데이터를 포함해서 전달하므로 전송하는 길이에 제한이 있음
- 브라우저 히스토리에 남고 캐시가 가능
 - 한 번 서버에 GET 요청을 한 적이 있다면 브라우저가 그 결과를 저장하고 이후 동일한 요청 처리 시 브라우저에 저장된 값 사용

POST

```
POST /user
body : {date : "example"}
Content-Type : "application/json"
```

- 서버에게 데이터 처리(주로 생성)을 요청
- 전달할 데이터를 **Body** 부분에 포함하여 통신
- **Request Body** 는 길이에 제한이 없기 때문에 대용량 데이터 전송 가능
- **Request Header** 의 **Content-Type** 에 요청 데이터의 Type을 표시해야 함
- 브라우저 히스토리에 남지 않고 캐시 불가능

PUT

```
PUT /user/1
body : {date : "update example"}
Content-Type : "application/json"
```

- 리소스를 대체할 때 사용
 - 리소스가 있으면 대체
 - 리소스가 없으면 생성
- 덮어쓰기와 같음
- 클라이언트가 리소스 위치를 알고 URI를 지정함 ➡ **POST** 와의 차이점

PATCH

- 리소스의 일부분을 수정할 때 사용
- 만약 서버에서 PATCH 메소드를 지원하지 않는 경우에는 **POST** 메소드 사용

DELETE

```
DELETE /user/1
```

- 리소스를 삭제할 때 사용

HTTP status code

- 클라이언트가 보낸 HTTP 요청에 대한 서버의 응답 코드
- 상태 코드를 통해 요청의 성공/실패 여부 판단
- 100-500번까지 총 5개의 클래스로 구분되어 HTTP 요청에 대한 상태 구분

- 5개의 클래스
 - 1xx (정보): 요청을 받았으며 작업을 계속 진행 중
 - 2xx (성공): 클라이언트가 요청한 동적을 성공적으로 처리
 - 3xx (리다이렉션): 요청을 완료하기 위해 추가 작업 조치 필요
 - 4xx (클라이언트 오류): 클라이언트의 요청에 문제 존재
 - 5xx (서버 오류): 서버가 유효한 요청의 수행을 실패
- 자주 등장하는 HTTP status code

status code	message	Description
200	OK	요청이 성공함 (ex. 잔액조회 성공)
201	Created	리소스 생성 성공 (ex. 게시글 작성 성공, 회원가입 성공)
400	Bad Request	데이터의 형식이 올바르지 않는 등 서버가 요청을 이해할 수 없음 (ex. 올바르지 않은 형식의 데이터 입력 등)
401	Unauthorized	인증되지 않은 상태에서 인증이 필요한 리소스에 접근 (ex. 로그인 전에 사용자 정보 요청 등)
403	Forbidden	인증된 상태에서 권한이 없는 리소스에 접근 (ex. 일반 유저가 관리자 메뉴 접근)
404	Not Found	요청한 route가 없음. 찾는 리소스가 없음. (ex. 존재하지 않는 route에 요청 등)
502	Bad Gateway	서버에서 예상치 못한 에러 발생 (ex. 예외처리를 하지 않은 오류 발생 등)

HTTPS (HyperText Transfer Protocol Secure)



인터넷 상에서 정보를 암호화하는 SSL(Secure Socket Layer) 프로토콜을 이용하여 웹브라우저(클라이언트)와 서버가 데이터를 주고 받는 통신 규약

- HTTP는 text 교환이므로, 누군가 네트워크에서 인터셉트할 경우 데이터 유출이 발생할 수 있는 보안 이슈 존재 ➡ 이러한 문제를 해결하기 위해 HTTP에 암호화를 추가한

HTTPS 사용

- SSL(보안 소켓 계층) 이나 TLS(전송 계층 보안) 프로토콜을 사용하여 데이터를 암호화
 - TLS 는 데이터 무결성을 제공하기 때문에 데이터가 전송 중에 수정, 손상되는 것을 방지
- 공개키 암호화 방식 사용
 - 공개키 암호화 : 비밀키를 공유하기 위해 사용
 - 제3자 인증 : 믿을 수 있는 인증기관에 등록된 인증서만 신뢰하는 것
 - 비밀키 암호화 : 통신하는 데이터를 암호화하는데 사용
- 초기에 클라이언트와 서버가 통신을 하며 암호화 키를 서로 안전하게 주고 받음. (SSL Handshake)
- 이 때 암호화 키값이 노출되지 않도록 안전하게 해주는 것이 https 서버 인증서

HTTPS 흐름

1. 클라이언트(브라우저)가 서버로 최초 연결 시도 (TCP Hand-Shaking)
2. 서버는 클라이언트에게 SSL 인증서(서버의 공개키 포함) 전송
3. 클라이언트가 인증서의 유효성 검사 → 데이터 암호화를 위한 대칭키 발급 (데이터 간의 교환에는 빠른 연산 속도가 필요하기 때문)
4. 클라이언트는 생성한 대칭키를 서버의 공개키로 암호화하여 서버에게 전송 (공개키 암호화 방식)
5. 서버는 자신의 개인키를 사용하여 받은 암호화된 대칭키를 복호화
6. 클라이언트와 서버 모두 같은 대칭키를 공유하게 됨
7. 클라이언트와 서버는 동일한 대칭키를 공유하므로 데이터를 전달할 때 이 대칭키를 이용하여 암호화/복호화를 진행하며 통신

“

클라이언트 : “안녕하세요? (경계중)”

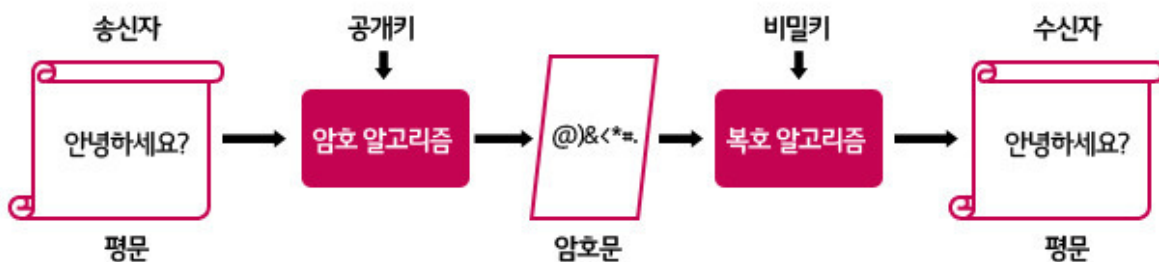
서버 : “(당당) 우리 SSL 인증된 사이트야. 공개키도 같이 던져줄게”

클라이언트 : “유효한거 맞아요? ... 맞네. 이제 말 놓자. 우리도 대칭키 하나 발급할건데, 앞으로 통신할 때 데이터들은 이걸로 보호해서 보낼거야. 너쪽 공개키로 한번 잠근거니까 너 개인키로 알아서 풀어”

서버 : “ㅇㅋ 내 개인키로 잘 풀리네. 확인”

대화 버전

📖 대칭키 암호화 vs 비대칭키 암호화



• 대칭키 암호화

- 클라이언트와 서버가 동일한 키를 사용해 암호화/복호화를 진행
- 공개키를 안전하게 교환하는 것이 핵심
- 키가 노출될 위험이 있지만 연산 속도가 빠름

• 비대칭키 암호화

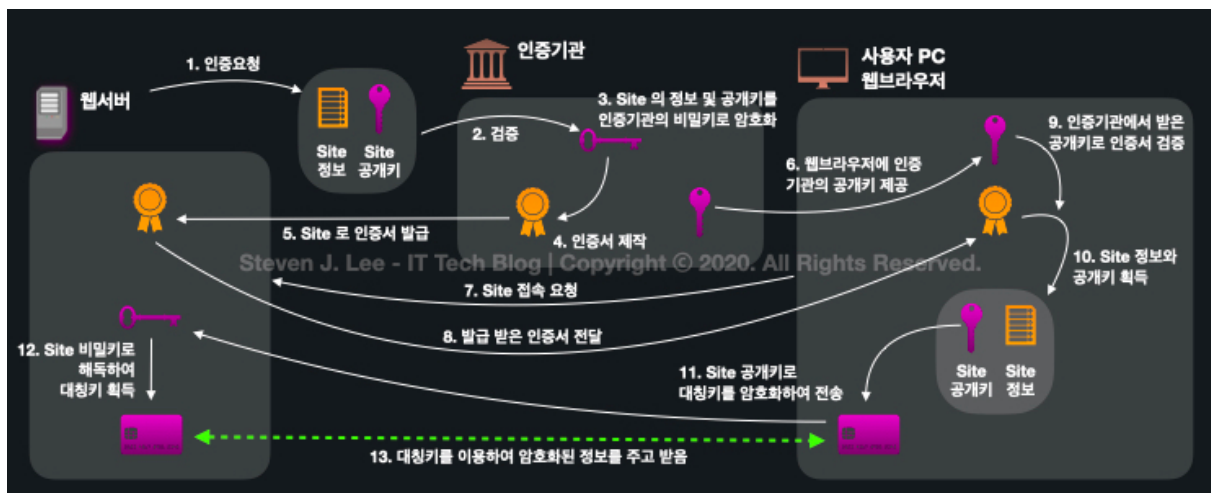
- 1개의 쌍으로 구성된 공개키와 개인키를 암호화/복호화 하는데 사용
- 키가 노출되어도 비교적 안전하지만 연산 속도가 느림
- 공개키로 암호화를 하면 개인키로만 복호화 가능
 - 개인키는 본인만 소유하므로, 내용 자체의 보안성 유지 가능

(출처: <https://zorbathegeek.tistory.com/30#HTTPS-1>)

📌 HTTPS 사이트를 만드는 과정과 SSL

1. 서버 개발자는 HTTPS를 적용하기 위해 공개키와 개인키를 생성. 이러한 키 쌍은 SSL/TLS 암호화 프로토콜에 사용됨.

2. 신뢰할 수 있는 CA(Certificate Authority) 기업을 선택하고, 해당 기관에 공개키 관리를 위한 계약을 진행.
3. 계약이 완료되면, CA는 서버의 정보(서버의 이름, 공개키, 공개키 암호화 방법 등)를 담은 SSL 인증서를 생성. 해당 인증서를 CA 기업의 개인키로 암호화해서 서버에게 제공.
4. 서버는 이 SSL 인증서를 클라이언트의 최초 연결 시도 때마다 클라이언트에 전송. 이를 통해 서버는 자신의 신원을 클라이언트에게 증명.
5. 클라이언트가 index.html 파일을 달라고 서버에 요청했다고 가정 → 처음 연결 시도에선 HTTPS 요청이 아니기 때문에 CA 기업이 자신의 개인키로 서버의 정보를 암호화한 인증서를 받음.
6. 브라우저는 CA 기업의 공개키로 인증서를 해독한 뒤 서버의 공개키를 얻음. CA 기업의 공개키는 브라우저가 이미 알고 있음.



SSL 통신과정

