HTTP & HTTPS

HTTP(HyperText Transfer Protocol)

- **HTTP는 클라이언트와 서버 사이에 이루어지는 요청(Request), 응답(Response) 프로토콜**을 의미한다.
  - **애플리케이션 레벨의 프로토콜**로, **WWW 상에서 정보를 주고받을 수 있다.**
  - HTML, 이미지, 동영상, 오디오 등 **종류를 가리지 않고 어떤 데이터든 전송**할 수 있다.

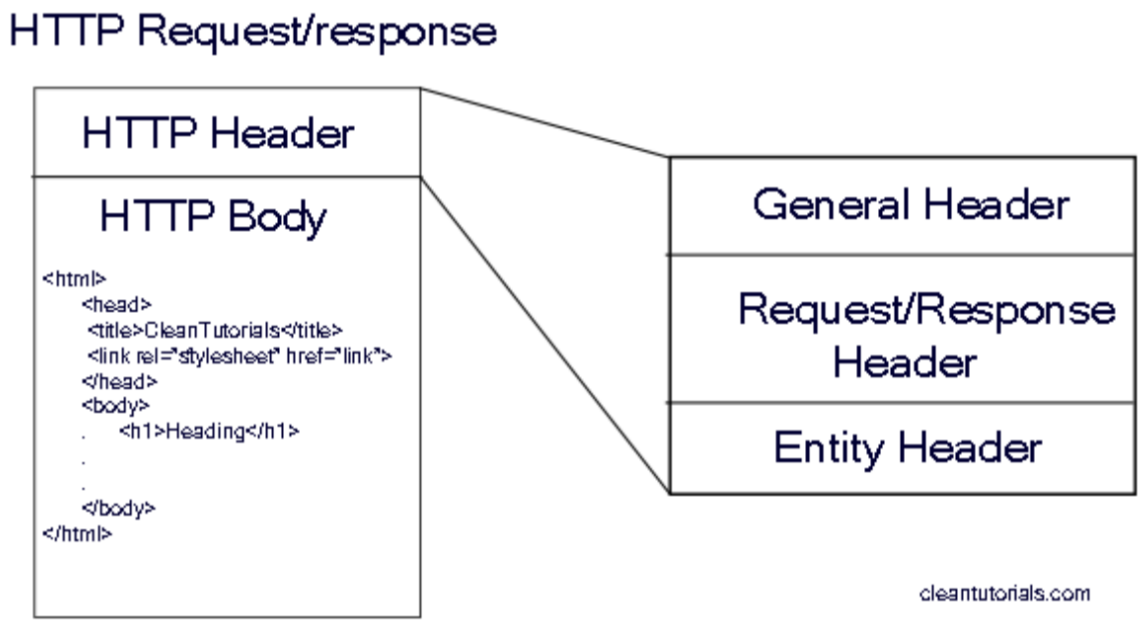
특징

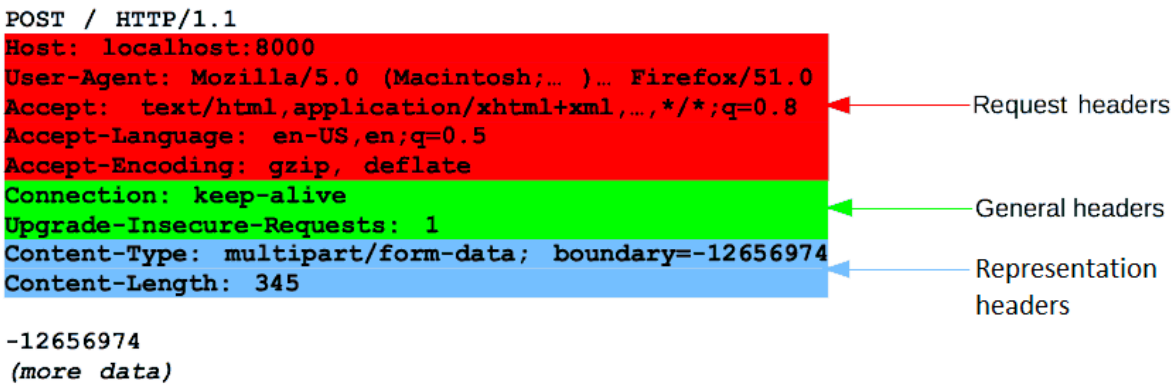
- **Connectless (비연결성)**
  - **클라이언트가 서버에 요청하고 응답을 받으면 바로 연결을 끊어버리는 성질**을 의미한다.
  - 기본적으로 **자원 하나에 대해 하나의 연결**을 만든다.
  - **불특정 다수 서비스에 적합하다.**
    - 수천명이 서비스를 사용하더라도 실제 서버에서 동시에 처리하는 요청은 수십개 이하로 적다.
    - **연결을 유지하기 위한 리소스를 줄여 더 많은 연결을 할 수 있다.**
  - 서버가 클라이언트를 기억하고 있지 않으므로, **동일 클라이언트의 요청에 대해 매번 새로운 연결을 시도 및 해제의 과정**을 거치며 **불필요한 오버헤드가 발생**한다.
    - **HTTP의 `KeepAlive` 속성을 통해 해결이 가능하다.**

        - `KeepAlive` 는 지정 시간동안 서버, 클라이언트 사이에 패킷 교환이 없을 경우 상대방에게 패킷을 보내며, 패킷에 반응이 없으면 접속을 끊는다.
        - 주기적으로 클라이언트 상태를 체크한다는 점에서 완벽한 해결책은 아니다.

- **Stateless (무상태성)**
    - 비연결성의 특성과 연계되어, **서버는 클라이언트의 이전 상태를 전혀 알수 없는 성질**을 의미한다.
    - 실제 서비스를 제공할 때 로그인 정보를 가지고 있는 경우와 같이 페이지를 벗어나도 계속해서 상태를 유지해야 할 필요성이 있다.
        - 이에 대한 대책으로 `Session`, `HTTP Cookie` 등이 있다.

구조





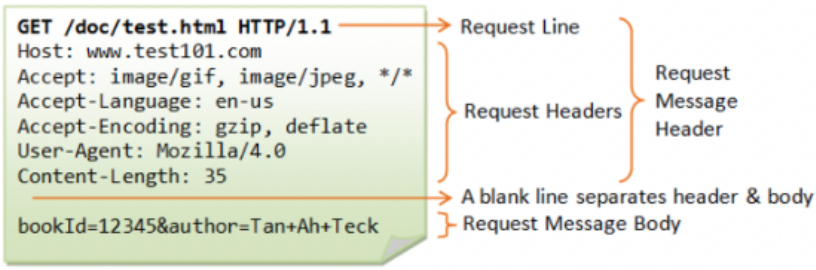
- HTTP 헤더와 바디로 구성되어 있고, HTTP 바디에는 **실제 통신과정에서 주고 받을 컨텐츠**가 포함되어 있다.
- 그 중 **HTTP 헤더는 해당 메시지가 제공하는 기능에 대한 최소한의 정보**를 담고 있다.
  - 불필요한 내용을 담으면 네트워크 전송 속도가 느려지므로 **프로토콜 설계 시 필요한 내용만** 담아야 한다.

General Header

- 전송되는 컨텐츠에 대한 정보보다는 **요청 및 응답이 이루어지는 날짜 및 시간등에 대한 정보가 포함**된다.
- **요청 헤더, 응답 헤더에 필수적으로 포함되는 헤더**이다.

- 구성 요소는 아래와 같다.
  - `Date`
    - 일반적인 HTTP 헤더는 만들어진 날짜와 시간을 포함하며, 요청 및 응답시에 자동 생성된다.
  - `Connection`
    - **현재 전송이 완료된 후 네트워크 접속의 유지 여부를 제어**한다.
    - 속성에는 `Keep-alive`, `close` 로 두 가지 옵션이 존재한다.
    - HTTP/1.1 의 경우 `Keep-alive` **로 연결을 유지하는 것이 기본값으로 지정**되어 있다.
      - `Keep-Alive: timeout=5, max=999`
  - `Cache-Control`
    - 캐싱의 허용 여부를 결정한다.

Request Header



- 요청 데이터는 `Header`, `Body` 로 구성된다.
  - `Header` 에는 요청과 요청 데이터에 대한 메타 정보들이 포함되어 있다.

- 구성 요소에 대한 자세한 설명은 아래와 같다.
  - `Host`
    - **서버 도메인 네임과 서버가 현재** `Listening` **중인 TCP 포트를 지정**한다.
    - 만약 포트가 보이지 않는다면, **요청 서버의 기본 포트가 지정됐음을 의미**한다.

        - HTTP 기본 포트 → 80
        - HTTPS 기본 포트 → 443
    - **호스트는 반드시 하나만 존재**해야 하며, **만약 없거나 여러개인 경우 `400` 상태 코드가 전송**된다.

- `User-Agent`
    - 현 사용자가 **어떤 클라이언트를 이용해 요청을 보냈는지 확인**할 수 있다.
    - `Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36`

- `Accept`
    - **요청 측(클라이언트)에서 처리 가능한 미디어 타입의 종류를 나열**한다.
    - `Accept: text/html` → HTML 데이터를 처리할 수 있다는 의미로 해석이 가능하다.
    - 물론, 응답하는 서버 측에서 해당 타입을 지원해야 의미가 있다.

- `Accept-Langauge`
    - **요청 측이 이해할 수 있는 문자 집합**을 서버에 알린다.
    - `Accept-Language: ko-KR`

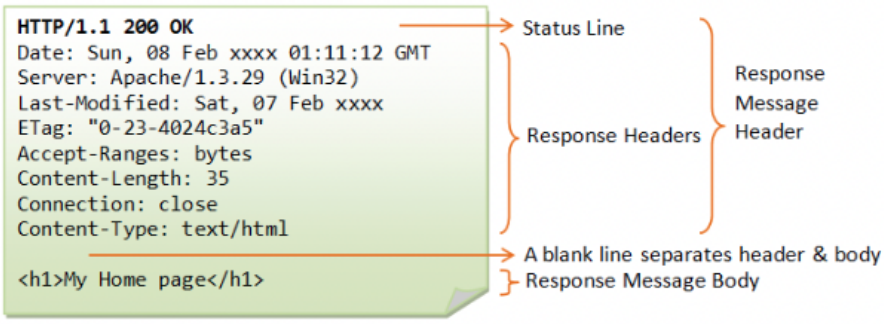
- `Accept-Encoding`
    - **요청 측에서 해석 가능한 압축 방식**을 지정한다.
    - `Accept-Encoding: gzip, deflate`

- `Authorization`
    - **사용자가 서버에 인가된 사용자임을 증명할 때 사용**된다.
    - 보통 **JWT**, **Bearer** 토큰을 서버로 보낼 때 사용하며 자격 미증명시 `401` 상태 코드가 전송된다.
    - `Authorization: <type> <credentials>`
        - `type` → 인증 타입
        - `credentials` → 사용자명, 비밀번호 등이 합쳐져 base64 로 인코딩 된 값
    - `Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l`

- `Origin`
    - **`POST` 와 같은 요청을 보낼 때 요청이 어느 주소에서 시작되었는지를 나타낸다.**
    - **만약 요청을 보낸 주소와 받는 주소가 다르면 `CORS` 문제가 발생한다.**
    - `Origin: https://developer.mozilla.org`

- `Referer`
    - **현재 페이지의 이전 페이지에 대한 주소**가 담겨있다.
    - 어떤 페이지에서 현재 페이지로 넘어왔는지 알 수 있어 페이지 통계를 낼 때 많이 사용한다.

Response Header



- 위치 혹은 서버 자체에 대한 정보와 같이 응답에 대한 부가적 정보를 포함하는 헤더이다.
  - 메세지의 컨텐츠와는 연관이 없다.

- 구성 요소는 아래와 같다.
  - `Access-Control-Allow-Origin`
    - 요청을 보내는 클라이언트 주소와 요청을 받는 백엔드 주소가 다르면 `CORS` 에러가 발생한다.
    - 해당 헤더에 클라이언트의 주소를 명시함으로서 에러를 방지할 수 있다.
    - `Access-Control-Allow-Origin: *`
      - 모든 주소에 CORS 요청을 허용하고 싶을 때 사용한다.
      - 보안이 취약해진다는 단점이 있다.

  - `Allow`
    - **서버에서 지원하는 메소드 집합을 나열**한다.
    - `GET` 메소드는 허용하지만 `POST` 메소드는 허용하지 않는 경우, `405` 상태 코드를 전송하며 헤더로 `GET` 을 보낸다.
    - `Allow : GET`
      - `GET` 요청만 받겠다는 의미이다.

  - `Content-Disposition`
    - 응답 내용에 담을 컨텐츠의 성향을 알리는 속성이다.
    - `Content-Disposition: inline`
      - 단순하게 웹페이지 화면에 표시된다.
    - `Content-Disposition: attachment; filename='filename.csv'`
      - 파일명과 함께 다운로드된다.

  - `Location`
    - **300번대 응답이나 201 응답일 때 어느 페이지로 이동할지 표시**한다.

        - `HTTP/1.1 302 Found Location: /`
            - 이 경우 브라우저가 `/` 주소로 리다이렉트된다.

    - `Content-Security-Policy` **(CSP)**
        - 해당 사이트에서 허용할 리소스의 종류(스크립트, 이미지, CSS 등)를 제어할 수 있다.
        - 신뢰할 수 없는 외부의 리소스가 해당 사이트의 웹 페이지에서 실행되지 않도록 제어할 수 있다. (XSS 공격 차단)
        - `Content-Security-Policy: default-src 'self'`
            - 가장 엄격한 설정으로 스크립트, 이미지, CSS 등의 리소스 출처가 자신의 사이트일 경우에만 실행이 허용된다.
        - `Content-Security-Policy: default-src 'none'; script-src https://cdn.mybank.net; style-src https://cdn.mybank.net; img-src https:/cdn.mybank.net; connect-src https://api.mybank.com; child-src 'self'`
            - 기본적으로 모든 리소스를 차단하므로 사이트는 아무 동작도 하지 않는다.
            - `script`, `stylesheet`, `img` 는 `cdn.mybank.net` 을 통해 다운로드 되는 리소스만 허용된다.
            - 외부 데이터 요청은 `api.mybank.com` 에서 응답 받은 내용만 가능하다.
            - `ifframe` 등의 하위 요소는 동일 도메인만 가능하다.

Entity Header

- **컨텐츠 길이, `MIME` 타입과 같이 엔티티 바디에 대한 자세한 정보를 포함하는 헤더**이다.
    - `Entity` → 컨텐츠, 본문, 리소스 등을 의미한다.

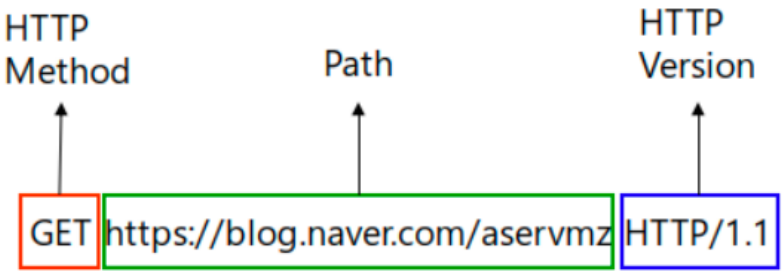
- 구성 요소는 아래와 같다.
    - `Content-Type`
        - 개체의 미디어 타입(MIME)과 문자열 인코딩을 지정하기 위해 사용한다.
        - `Content-Type: text/html; charset=utf-8`
        - **타입을 지정하지 않는다면 수신 측에서는 모든 데이터를 단순 텍스트 데이터로 인식한다.**
            - 텍스트를 분석하는 추가적 코딩을 원치 않는다면 필수적으로 명시하여야 한다.

    - `Content-Language`
        - 사용자들에게 언어를 설명하기 위해 사용되며, 선호 언어에 따라 사용자를 구분할 수 있다.
        - 미지정 시 모든 사람들에게 공개된 정보라고 간주한다.
        - `Content-Language: en-US`

    - `Content-Encoding`
        - 미디어 타입을 압축하기 위해 사용되며, 사용자는 값이 어떤 방식으로 인코딩되는지 확인할 수 있다.
        - `Content-Encoding: gzip` → 브라우저가 스스로 해제해 사용 가능하다.

- `Content-Length`
  - **헤더 + 바디의 크기**를 나타내며, 메시지 크기에 따라 자동 생성된다.

HTTP 요청 메소드



- **HTTP에서 요청 전송 시 HTTP 메소드를 포함하여 전송하게 된다.**
  - 어떤 기능을 수행할 것인지 부가적으로 설명하는 역할을 맡는다.

- **멱등성(Idempotent)**
  - **같은 행위를 여러 번 수행하더라도 결과가 같은 경우**를 의미한다.
    - HTTP 메소드에서의 **결과**가 의미하는 것은, 응답 상태 코드가 아닌 **서버의 상태**이다.
    - **똑같은 요청을 했을 때 상태 코드가 변경되더라도 서버의 상태가 항상 같은 상태라면 멱등성이 있다고 판단한다.**

GET	- 특정 리소스를 검색 및 취득 시 사용한다. - 요청 시 전송 데이터를 `URI` 에 쿼리 파라미터 형태로 포함시킨다.
POST	- 새로운 리소스를 생성할 때 사용한다. - 요청 시 전송 데이터를 `HTTP Body` 에 포함시킨다. - 리소스 신규 생성 시 해당 리소스에 대한 `URI` 를 `Location` 헤더에 포함시켜 응답한다. **- 멱등성을 보장하지 않는다.**
DELETE	- 특정 리소스를 제거할 때 사용한다.
PUT	- 특정 리소스 내용을 수정할 때 사용한다.
PATCH	- 특정 리소스를 부분적으로 수정할 때 사용한다. **- 멱등성을 보장하지 않는다.**
HEAD	- 특정 리소스를 `GET` 메소드로 요청했을 때 반환되는 헤더를 요청할 때 사용한다. - HTTP 헤더만을 요구하므로 응답에 `HTTP Body` 가 존재하지 않는다.
OPTION	- 서버측이 제공하는 HTTP 메소드의 종류에 대해 질의할 때 사용한다. - 응답의 `Allow` 헤더에 사용 가능한 HTTP 메소드 종류가 포함된다.
TRACE	- 요청 리소스가 수신되는 경로를 확인할 때 사용한다.
CONNECT	- 요청 리소스에 대한 양방향 연결을 시작하기 위해 사용한다. - 원하는 목적지와의 TCP 연결을 HTTP 프록시 서버에 요청한다.

- **`GET` 메소드는 항상 멱등성이 있다고 할 수 있을까?**
  - 기본적으로는 데이터를 **단순 조회하는 메소드**이기 때문에 상태 변경이 일어나지 않는 것은 당연하다.
  - 하지만 `GET` **요청 시 게시글 조회수를 늘린다고 가정**했을 때, 같은 요청을 보내더라도 서버의 상태는 변경된다.

⇒ **멱등성을 가지지 않는 것이며 HTTP 스펙에 부합하지 않은 구현이라고 볼 수 있다.**

- `POST` **메소드는 왜 멱등성이 없을까?**
  - `CRUD` **관점에서 볼 때 일반적으로 새로운 자원을 생성하는 역할**을 하기 때문이다.
  - **같은 요청을 보낼 때 매번 새로운 자원이 생겨날 수 있으며, 이는 서버 상태의 변경을 의미**한다.

- `DELETE` **메소드는 왜 멱등성이 있을까?**
  - **명확하게 지정한 리소스를 삭제**하는 메소드이기 때문에, 여러 번 요청하더라도 서버의 상태는 동일하다.
  - 멱등성을 보장하라는 것이 명시되어 있기 때문에, **호출 시 정확한 식별자를 통해 리소스를 지정**해야 한다.
  - **만약 가장 마지막 게시글을 삭제하라는 API**가 있다면, **매 번 마지막 게시글을 삭제하기 때문에 멱등성을 보장하지 않는다.**
    - `DELETE /post/last`

      ⇒ **이러한 경우 멱등성을 미보장하는** `POST` **메소드를 사용하는 것이 옳다.**

- `PUT` **메소드는 왜 멱등성이 있을까?**
  - **대상 자원이 있다면 데이터만 덮어쓰기 때문**에, 몇 번을 하더라도 서버의 상태는 같다.
  - 단, HTTP 스펙에서 정의된 `PUT` **의 응답 코드**는 다음과 같다.
    - 새 데이터를 생성 시 `201` 상태 코드를 응답한다.
    - 기존 데이터를 덮어 쓴 경우 `200` 혹은 `204` 상태 코드를 응답한다.

      ⇒ **상태 코드가 동일하지 않더라도 서버의 상태는 동일하므로 멱등성을 가진다.**

- `PATCH` **메소드는 왜 멱등성이 없을까?**
  - **결정적인 이유는 HTTP 스펙 상 구현 방법에 제한이 없기 때문이다.**
  - `PUT` 처럼 기존 데이터에 데이터를 덮어씌움으로서 멱등성 있게 구현할 수도 있다.
  - 그러나, `PATCH` 를 통해 사용자 나이를 1살 늘려달라는 식으로 구현을 했다면 멱등성이 없다고 볼 수 있다.

    ⇒ **즉, API를 구현하는 개발자의 성향에 따라 멱등성을 보장할 수도, 보장하지 않을 수도 있는 것이다.**

  ▼ **멱등하지 않은** `PATCH` **예제**

```
기존의 리소스
{
  id: 1,
  name: "김철수",
  age: 15
}

PATCH /users/1
{
  age: {
    type: $inc,
    value: 1
  }
}

변경된 리소스
{
  id: 1,
  name: "김철수",
  age: 16
}
```

상태 코드 (Status code)

- **요청에 대한 응답의 결과는 상태 코드를 통해 전달된다.**
- 상태 코드는 세 자리 숫자로 이루어져 있으며, 크게 다섯개의 클래스로 분류된다.

- `1XX` → 조건부 응답
  - HTTP 요청이 수신되었으며 다음 처리를 계속해서 진행함을 의미한다.

- `2XX` → **성공**
  - **요청을 성공적으로 처리했음을 의미**한다.
  - `200, 성공` → 서버에서 요청을 성공적으로 처리했다.
  - `201, 생성됨` → 요청을 성공적으로 처리했으며, 새로운 리소스를 생성했다.
  - `202, 허용됨` → 요청을 받았으나 완전히 처리하지는 못했다.
  - `204, 컨텐츠 없음` → 요청을 성공적으로 처리했으며, 요청과 관련된 컨텐츠 또한 존재하지 않음을 의미한다.
  - `206, 일부 컨텐츠` → `GET` 요청의 일부만 성공적으로 처리했다.

- `3XX` → **리다이렉션**
  - **요청 완료를 위해 추가적 처리나 동작이 필요함을 의미**한다.
  - **다른 `URL`을 통해 해당 리소스에 접근해야 하는 경우 등장**한다.

  - `301, 영구이동` → 요청 페이지가 다른 위치로 영구 이동했다.
    - `301` 을 받으면 `Location` 헤더에 담겨있는 URL로 자동 리다이렉션한다.

  - `302, 임시이동` → 요청 페이지가 다른 위치로 임시 이동했고, 클라이언트는 여전히 현 페이지를 요청해야 한다.

  - `304, 수정되지 않음` → 마지막 요청 이후 요청 페이지가 수정되지 않았다.
    - `304` 를 받으면 클라이언트는 굳이 서버에게 리소스를 재응답받아야 할 필요가 없기 때문에 **캐싱해놓았던 리소스를 사용**한다. `(Conditional Get)`
    - 자신이 캐싱해놓았던 리소스를 사용하는 것이므로 이 역시 캐싱된 리소스로 리다이렉션되었다고 한다.
    - 캐싱된 리소스가 없는 경우 빈 화면 혹은 에러 화면을 띄운다.

- `4XX` → **요청 오류**
  - **클라이언트의 요청에 오류가 있음을 의미**한다.

  - `400, 잘못된 요청` → 잘못된 문법으로 인해 서버가 요청을 이해할 수 없음을 의미한다.

    - `401, 권한 없음` → **인증이 필요한 리소스를 요청**하는 경우 인증이 필요함을 알린다.
        - 로그인이 필요한 API를 비로그인 사용자가 호출하는 경우 많이 사용된다.
        - 클라이언트에서 `401` 을 받으면 로그인이 필요한 것으로 판단하고 로그인 페이지로 리다이렉션 하기도 한다.

    - `403, 금지` → 클라이언트가 접근이 금지된 리소스를 요청했음을 의미한다.
        - 해당 리소스를 사용하는 것은 인증 여부와 관계없이 **무조건적으로 금지**라는 것을 의미한다.
        - HTTPS 프로토콜로 접근해야 하는 리소스에 HTTP 프로토콜을 사용해 접근할 경우 주로 사용한다.

    - `404, 찾을 수 없음` → 요청 리소스가 존재하지 않음을 의미한다.
    - `405, 허용하지 않는 방법` → 현재 리소스에 맞지 않는 메소드를 사용했음을 의미한다.

    - `406, 허용되지 않음` → **서버 주도 컨텐츠 협상**을 진행했음에도 알맞은 컨텐츠 타입이 없음을 의미한다.
        - 클라이언트가 보낸 `Accept` 헤더와 맞는 `Content-Type` 을 모두 찾아보았음에도 알맞은 리소스가 없는 경우 사용한다.

    - `408, 요청시간 초과` → 클라이언트, 서버 간 연결은 성사됐으나 요청 본문이 서버에 도착하지 않는 상황을 의미한다.
        - 서버가 아무리 기다려도 클라이언트가 보낸 요청을 받지 못하는 경우 발생한다.

    - `429, 너무 많은 요청` → 클라이언트가 서버에 너무 요청을 많이 보내는 경우 발생한다.
        - 너무 짧은 시간에 빠르게 요청을 보내는 경우나, 유료 API를 사용할 때 사용할 수 있는 API 요청 횟수를 초과하는 경우 주로 사용한다.
        - `429` 와 함께 응답 헤더의 `Retry-After` 를 이용해 지정한 시간 이후 재요청하라는 안내를 전달할 수 있다.

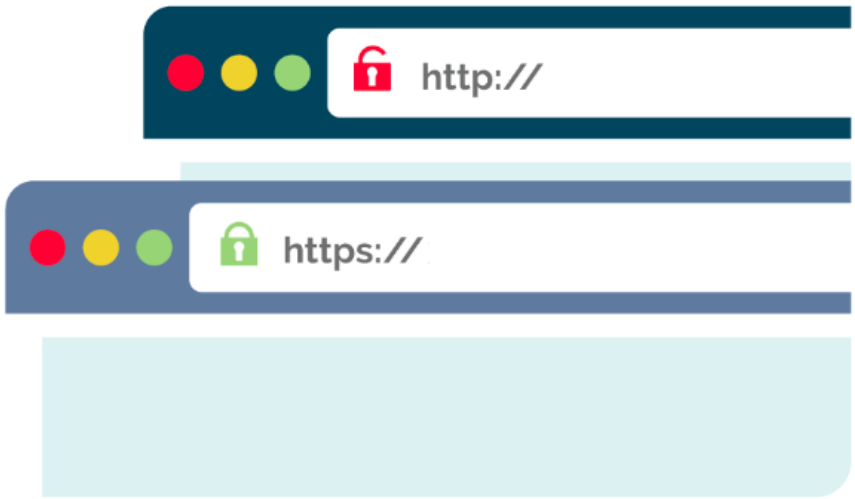
- `5XX` → **서버 내부 오류**
    - **서버 내부에서 오류가 발생한 경우를 의미**한다.

    - `500, 내부 서버 오류` → 백엔드 어플리케이션 내부에서 알 수 없는 에러가 발생한 경우를 의미한다.
        - 핸들링되지 않은 에러가 주로 발생하므로, **에러 원인을 클라이언트에게 알려주지 않는다.**

    - `502, 게이트웨이 불량` → 게이트웨이가 연결된 서버로부터 잘못된 응답을 받았을 때 사용된다.

    - `503, 일시적으로 서비스 이용 불가` → 서버가 요청을 처리할 준비가 되지 않았음을 의미한다.
        - **일시적 상황**을 강조하는 의미가 크며, 일반적으로 서버에 부하가 심해 현재 요청을 처리할 수 있는 여유가 없는 경우에 많이 사용된다.
        - `403` 과 마찬가지로 `Retry-After` 헤더를 이용해 지정 시간 이후 재요청하라는 안내를 전달할 수 있다.

HTTPS(Hypertext Transfer Protocol Secure)

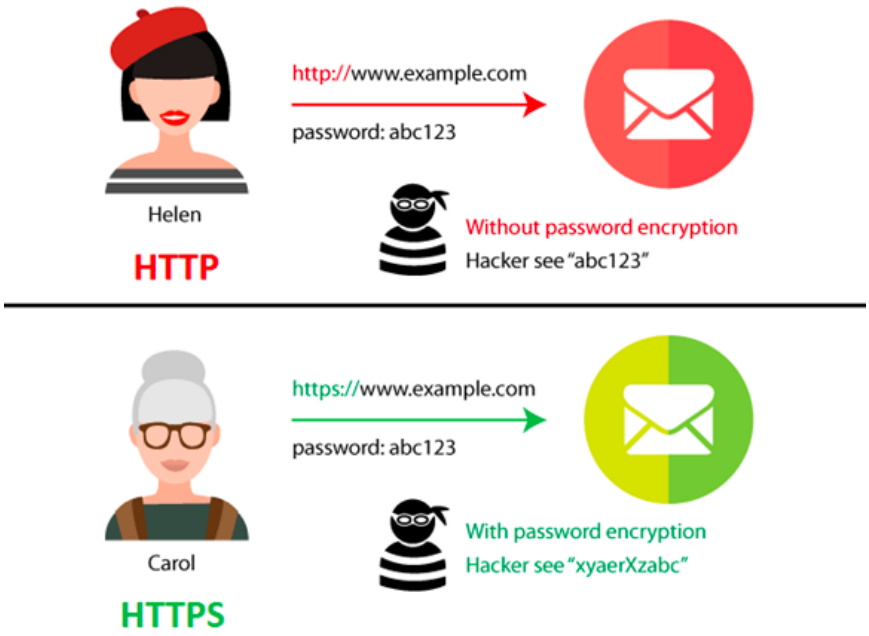


- HTTP는 정보를 텍스트로 주고 받기 때문에 네트워크에서 전송 신호를 인터셉트 하는 경우 **원하지 않는 데이터 유출이 발생**할 수 있다.

  ⇒ **이러한 보안 취약점을 해결하기 위해 HTTPS 프로토콜이 대두되었다.**

사용 이유

1. **어떤 웹사이트에 보내는 정보를 외부인이 살펴볼 수 없도록 모든 통신 내용이 암호화된다.**
   - HTTP 프로토콜을 통해 데이터를 전송하면, 데이터가 변형되지 않은 채 그대로 전송된다.
   - 만약 외부인이 중간에 데이터를 볼 수 있다면 개인정보가 그대로 탈취될 것이다.



- HTTPS는 데이터를 변형하여 전송하기 때문에 누군가 데이터를 보더라도 정보를 알아낼 수 없도록 한다.

2. **접속한 사이트가 신뢰할 수 있는 사이트인지에 대한 정보를 준다.**
   - **기관으로 검증된 사이트만** 주소에 HTTPS 프로토콜 사용이 허용된다.
   - HTTP만 사용하는 사이트는 별도로 안전하지 않다는 표시를 해준다.

동작 원리

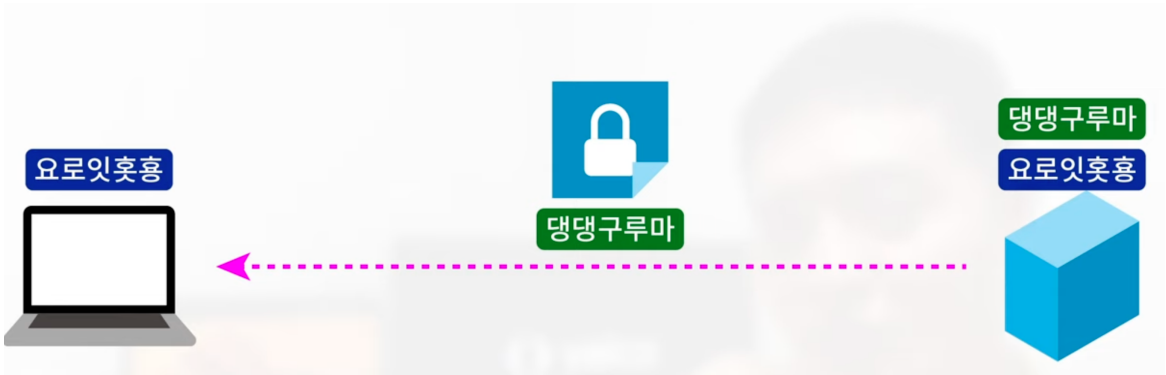
- **SSL** 혹은 **TLS** 프로토콜을 통해 데이터를 암호화한다.
  - **TLS(Transport Layer Security)** → 과거 SSL에서 발전하며 이름이 변경된 것을 의미하나, 현재도 SSL(Secure Socket Layer) 이라는 명칭이 많이 사용되고 있다.
  - **웹사이트가 SSL 혹은 TLS 프로토콜로 보호되는 경우 HTTPS 가 URL에 표시된다.**

- **SSL**은 **공개키 암호화 방식과 대칭키 암호화 방식을 함께 사용**한다.
  - SSL 적용을 위해 인증서를 발급받아 서버에 적용시켜야 한다.
  - 인증서는 사용자 입장에서 접속한 서버가 진입하고자 했던 서버가 맞는지 보장해주는 역할을 한다.
  - 그리고, 이런 인증서를 발급하는 기관을 CA(Certificate Authority) 라고 부른다.

- **CA란?**
  - 공개키와 공개 DNS명의 연결을 보장해주는, 공인된 민간 기업들을 의미한다.
  - 크롬, 사파리, 파이어폭스와 같은 브라우저에 CA들의 목록이 내장되어 있다.

동작 과정

1. **클라이언트는 현재 서버를 신뢰하지 못하는 상태**이기 때문에, 핸드셰이크를 통한 탐색 과정을 시작한다.



   - **클라이언트가 랜덤 데이터를 생성하여 서버에 전송**한다.
   - **서버측에서도 답변으로 랜덤 데이터와 해당 서버의 인증서를 함께 클라이언트에 전송**한다.

2. **클라이언트측은 받은 인증서를 신뢰할 수 있는지 검증한다.**

- 브라우저에 내장된 **CA들의 정보**를 통해 **공개키 방식으로 인증서 검증을 시작**한다.
- **CA 인증을 받은 인증서의 경우 해당 CA의 개인키로 암호화가 되어있다.**
- **신뢰할 수 있는 인증서**라면 브라우저에 저장된 **CA의 공개키로 복호화**할 수 있다.
- 만약 CA 인증을 받지 않은 인증서라면, 브라우저 주소창에 **신뢰할 수 없다는 메시지가 등록**된다.

3. **인증서를 복호화**했다면, **서버와 데이터를 주고받기 위한 준비를 시작한다.**
   - **성공적으로 복호화된 인증서에는 서버의 공개키가 포함**되어 있다.
   - **데이터를 주고 받는 과정은, 대칭키 방식과 공개키 방식이 혼합되어 사용된다.**

   - **우선, 대칭키를 공유하기 위해 공개키 방식을 사용한다.**
     - 현재 클라이언트측은 1번 과정에서 자신이 생성한 랜덤 데이터와, 서버에게 전송받은 랜덤 데이터를 가지고 있다.
     - **두 랜덤 데이터를 혼합해 임시키를 생성**한다.

- **임시 키는 서버의 공개키로 암호화**되어 서버로 보내지고, **서버는 개인키를 통해 이를 복호화**한다.
- 이후 각자 일련의 과정을 거치고 나면 **서버와 클라이언트는 동일한 대칭키를 가지게 된다.**

- 동일한 대칭키를 서버와 클라이언트가 갖고 있기 때문에, **대칭키 방식으로 데이터를 주고 받는다.**

- **당연하게도 서버, 클라이언트 외엔 대칭키를 가지고 있지 않기 때문에 악의적으로 데이터를 훔칠 수 없다.**
- 세션이 종료되면 대칭키는 폐기된다.

🤔 **데이터를 주고 받을 때 대칭키 방식과 공개키 방식을 혼합해서 사용하는 이유는 뭘까?**

- 공개키 방식으로 데이터를 일일이 암호화 및 복호화하는 것은 컴퓨터에 큰 부담을 주기 때문이다.

장점

- **웹사이트의 무결성을 보호**해준다.
  - 웹 사이트와 사용자 브라우저 사이 통신을 침입자가 건드리지 못하도록 한다.
  - **즉, 안전하다.**
- 네이버, 다음, 구글과 같은 검색 사이트의 검색 엔진은 HTTPS 프로토콜을 사용한 사이트를 우선적으로 상위목록에 표시되게끔 지정한다.

단점

- 설치 및 인증서를 유지하는 과정에서 **추가 비용이 발생**한다.
- **HTTP 프로토콜에 비해 속도가 느리다.**
  - **민감한 정보를 다루지 않는 웹사이트의 경우 HTTP 프로토콜을 사용한다.**
- 인터넷 연결이 끊긴 경우 **재인증 시간이 소요**된다.
- 신뢰할 수 없는 CA 기업이 인증서를 발급할 수 있으므로 매번 안전한 것이라고 볼 수 없다.

References

- https://opentutorials.org/course/3385
- https://ko.wikipedia.org/wiki/HTTP
- https://www.joinc.co.kr/w/Site/Network_Programing/AdvancedComm/HTTP
- https://victorydntmd.tistory.com/286
- https://codify.tistory.com/91
- https://www.zerocho.com/category/HTTP/post/5b4c4e3efc5052001b4f519b\
- https://blog.naver.com/PostView.naver?blogId=aservmz&logNo=222301982303&from=search&redirect=Log&widgetTypeCall=true&directAccess=false
- https://evan-moon.github.io/2020/03/15/about-http-status-code/
- https://rachel-kwak.github.io/2021/03/08/HTTPS.html
- https://www.youtube.com/watch?v=H6lpFRpyl14