HTTP

HTTP 프로토콜?

Hypertext Transfer Protocol 의 약자로 웹 통신 프로토콜입니다. 여기서 프로토콜이란 상호 간의 규칙을 의미하며 그렇기에 웹 통신 프로토콜이란 **특정 기기 간에 데이터를 주고 받는데 필요한 규칙** 정도로 해석됩니다.

특징

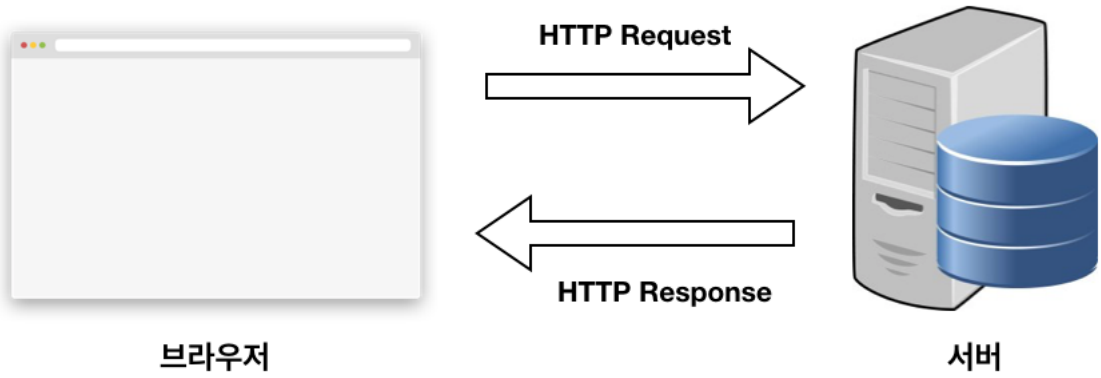
HTTP 프로토콜은 **상태가 없는(stateless) 프로토콜**입니다. 여기서 stateless는 데이터를 주고 받기 위한 각각의 Request가 서로 독립적으로 관리가 된다는 뜻으로, 좀 더 쉽게는 **이전 데이터 요청과 다음 데이터 요청이 관련이 없다**는 뜻입니다. 예를 들어, 로그인 요청이 회원가입 요청에 관여하지 않죠.

이러한 특징 덕에 서버는 **세션과 같은 별도의 추가 정보를 관리하지 않아도 되고, 다수의 요청 처리 및 서부의 부하를 줄일 수 있습니다.**

HTTP 프로토콜은 일반적으로 **TCP/IP** 통신 위에서 동작하며 기본 포트는 80번입니다.

HTTP Request & Response

HTTP 프로토콜로 데이터를 주고받기 위해서는 요청(Request)을 보내고 응답(Response)를 받아야 합니다.

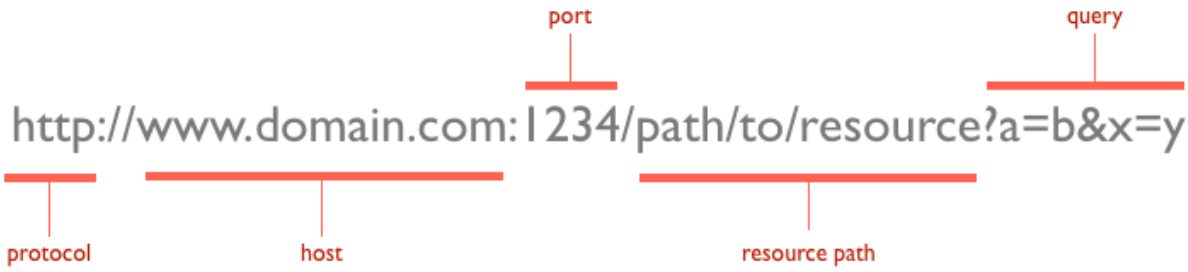


클라이언트 측에서 서버로 요청을 보내면 서버는 그에 대한 동작을 실행하고 응답을 클라이언트로 보냅니다. 그리고 클라이언트는 해당 응답을 가지고 브라우저를 동작시켜줍니다.

예를 들어 클라이언트 측에서 서버에 로그인 요청을 보내면 서버는 클라이언트가 보낸 정보(이메일, 패스워드 등등)를 가지고 해당 유저가 데이터베이스에 있다면 성공을, 없다면 실패를 클라이언트에게 응답으로 보내줍니다. 클라이언트는 성공을 받았다면 로그인이 된 이후의 동작을 실행하겠죠.

URL

URL(Uniform Resource Locators)은 서버에 자원을 요청하기 위해 입력하는 영문 주소입니다. 구조는 아래와 같습니다.



HTTP 요청 메서드

URL을 이용해 서버에 특정 데이터를 요청할 수 있습니다. 이때, 특정 동작을 수행하고 싶으면 **HTTP 요청 메서드**를 이용합니다.

- **GET** : 존재하는 자원에 대한 **요청**
- **POST** : 새로운 자원 **생성**
- **PUT** : 존재하는 자원에 대한 **변경**
- **DELETE** : 존재하는 자원에 대한 **삭제**

즉, 우리가 CRUD라고 부르는 Create, Read, Update, Delete는 각각 POST, GET, PUT, DELETE 에 해당됩니다.

참고로 POST 메서드로 PUT, DELETE도 사용이 가능하지만 특정한 상황을 제외하고는 용도에 맞게 사용하는 것이 좋습니다.

기타 요청 메서드는 아래와 같습니다.

- HEAD : 서버 헤더 정보를 획득. GET과 비슷하지만 Response Body를 반환하지 않는다.
- OPTIONS : 서버 옵션들을 확인하기 위한 요청으로 CORS에 사용

HTTP 상태 코드

언젠가 어떤 페이지에 들어갔는데 404 Not Found 라는 문구를 본 적 있을겁니다. 해당 페이지가 서버에 없다는 에러 메세지인데 여기서 404 코드를 HTTP 상태 코드라고 부릅니다.

주로 200번대를 성공, 400번대를 실패로 나누고 200~500번대까지 다양하게 상태 코드가 존재합니다.

2xx - 성공

200 번대의 상태 코드는 대부분 성공을 의미합니다.

- 200 : GET 요청 성공
- 204 : No Content. 성공했으나 응답 본문에 데이터 없음
- 205 : Reset Content : 성공했으나 클라이언트 화면을 새로 고침하도록 권고
- 206 : Partial Content : 성공했으나 일부 범위의 데이터만 반환

3xx - 리다이렉션

300번대 상태 코드는 대부분 클라이언트가 이전 주소로 데이터를 요청하여 서버에서 새 URL로 리다이렉트를 유도하는 경우입니다.

- 301 : Moved Permanently, 요청한 자원이 새 URL에 존재
- 303 : See Other, 요청한 자원이 임시 주소에 존재
- 304 : Not Modified, 요청한 자원이 변경되지 않았으므로 클라이언트에서 캐싱된 자원을 사용하도록 권고

4xx - 클라이언트 에러

400번대 상태 코드는 대부분 클라이언트 코드가 잘못된 경우입니다.

- 400 : Bad Request, 잘못된 요청
- 401 : Unauthorized, 권한 없이 요청. Authorization 헤더가 잘못됨
- 403 : Forbidden, 서버에서 해당 자원에 대해 접근 금지
- 405 : Method Not Allowed, 허용되지 않은 요청 메서드
- 409 : Conflict, 최신 자원이 아닌데 업데이트하는 경우

5xx - 서버 에러

500번대 상태 코드는 서버 쪽에서 오류가 난 경우입니다.

- 501 : Not Implemented, 요청한 동작에 대해 서버가 수행할 수 없는 경우

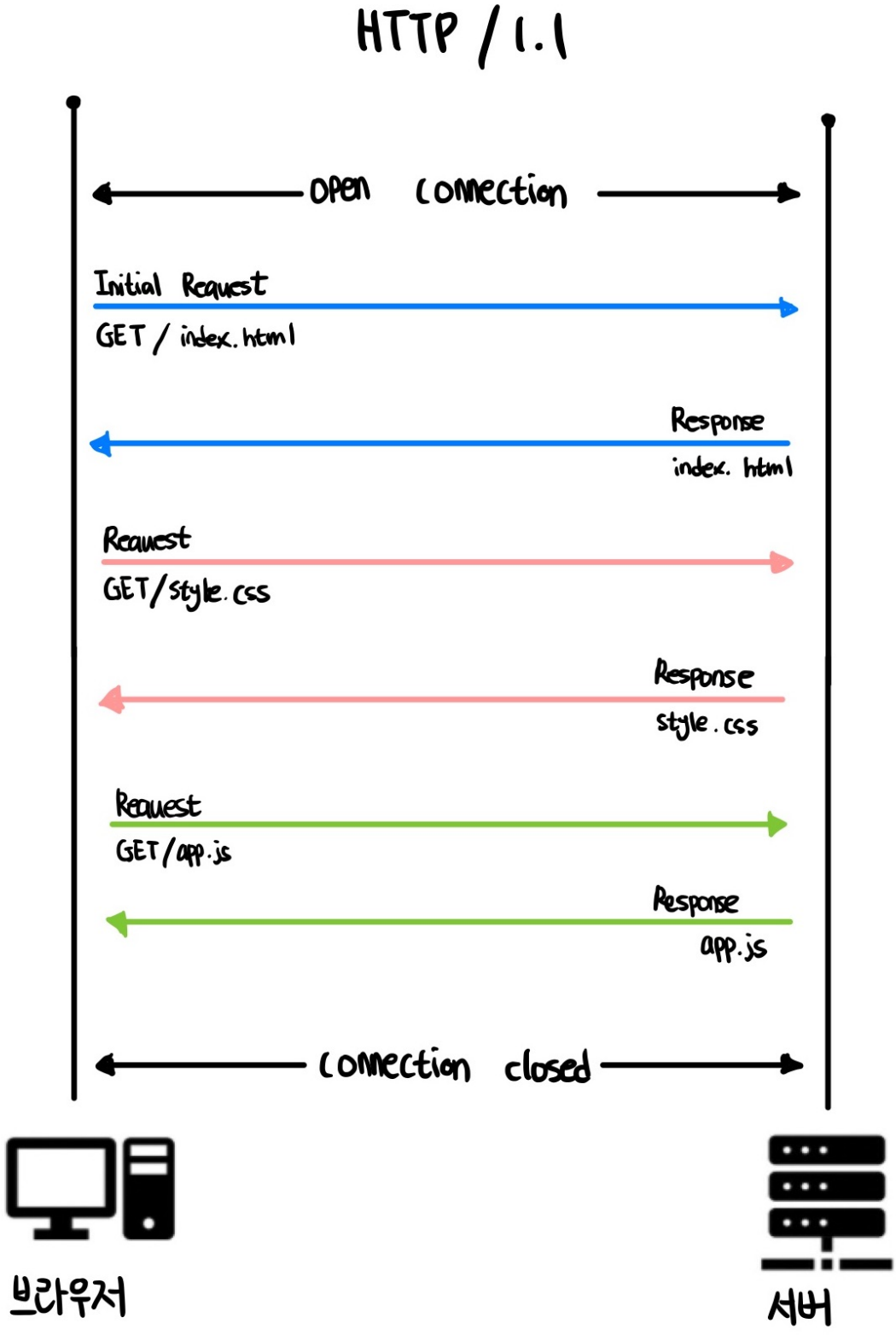
- 503 : Service Unavailable, 서버가 과부하 또는 유지 보수로 내려간 경우

HTTP/1.1, HTTP/2.0

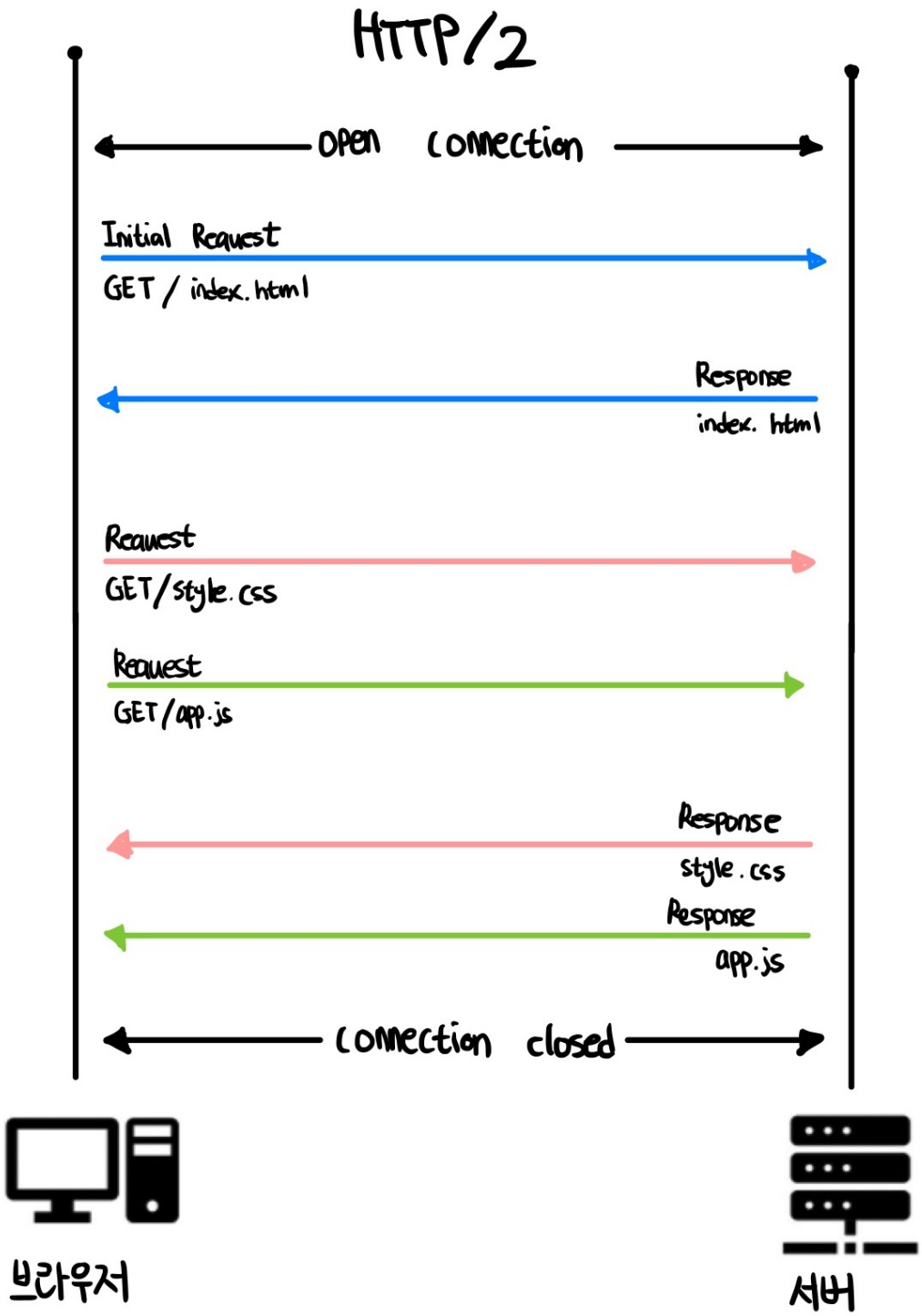
HTTP/1.1은 기본적으로 **커넥션 당 하나의 요청과 응답**만 처리합니다. 즉, 여러 개의 요청 및 응답을 한 번에 전송할 수 없습니다.

만약 한 페이지를 위해 여러개의 동작이 페이지 로드 시 필요하다면, 아래와 같이 요청, 응답을 한 사이클로 여러번 통신이 이루어져야 합니다.

만약 동작의 개수가 증가하면 증가할수록 시간도 오래 걸릴수 밖에 없습니다.


HTTP / 1.1
open connection
Initial Request
GET / index.html
Response
index. html
Request
GET/style.css
Response
style.css
Request
GET/app.js
Response
app.js
connection closed
브라우저
서버

반면 HTTP/2.0은 **커넥션 당 여러 개의 요청과 응답**이 가능합니다. 이러한 특징 때문에 1.1 보다 50% 정도 빠르다고 알려져 있습니다.



💡 HTTPS vs HTTP

HTTPS는 HTTP + 데이터 암호화 입니다. HTTP와 다르게 443번 포트를 사용하며, 네트워크 상에서 중간에 제3자가 정보를 볼 수 없도록 **공개키 암호화**를 지원합니다.

HTTP 형식으로 정보를 보낼 경우, 아래와 같이 입력한 형태 그대로 데이터가 보내지게 됩니다.

```
id: email@naver.com
pw: password
```

당연히 보안상으로 좋을 수가 없는 구조입니다.

하지만 HTTPS를 사용한다면 요청을 보낼 때 응답을 해주는 서버만 알아볼 수 있도록 **정보를 암호화**해서 보내게 됩니다.

```
id: ^!@$!A_!@E!@#(_a*&@)
pw: !@#%_+#$%_!#@!$
```

물론 위와 같이 암호화가 되지는 않겠지만 특정한 방식을 통해 데이터를 암호화시켜 서버에 전송합니다. 당연히 HTTP 형식보다 훨씬 보안상 이점이 큽니다.

또한 HTTPS는 기관으로부터 검증된 사이트만 주소에 HTTPS 사용이 허가되기 때문에, 내가 접속한 사이트가 상대적으로 안전한 주소를 가졌음을 증명해주기도 합다.

💡 공개키(비대칭키) 암호화 방식

위에서 HTTPS 는 공개키 암호화 방식을 이용해 데이터를 암호화 시킨다고 했습니다. 조금 더 그 방식을 자세히 알아보겠습니다.

HTTPS는 **공개키, 개인키, 대칭키**를 가지고 데이터를 암호화합니다.

- **공개키** : 모두에게 공개된, 또는 공개가 가능한 키
- **개인키** : 나만 가지고 있는, 나만 알고 있어야 하는 키
- **대칭키** : 동일한 키로 암호화, 복호화가 가능하다. 매번 랜덤으로 생성되기 때문에 안전하며 공개키보다 빠르다.

공개키와 개인키로 암호화하면 다음과 같은 효과를 얻을 수 있습니다.

- **공개키 암호화** : 공개키로 암호화를 하면 개인키로만 복호화 할 수 있다. → 개인키는 나만 가지고 있으므로, 나만 볼 수 있다.
- **개인키 암호화** : 개인키로 암호화 하면 공개키로만 복호화할 수 있다. → 공개키는 모두가 가지고 있으므로, 내가 인증한 정보임을 알려 신뢰성을 보장할 수 있다.

HTTPS를 사용하기 위해서는 위의 CA에 공개키를 전송하여 인증서를 발급받아야 합니다.

여기까지 생각하고 좀 더 자세하게 HTTPS의 동작 과정을 알아보겠습니다.

1. 서버는 공개키와 개인키를 만들고, 신뢰할 수 있는 인증기관인 CA에 자신의 정보와 공개키를 관리해 달라고 계약하고 돈을 지불한다.
2. 이때, CA 또한 CA만의 공개키와 개인키가 존재하며 CA와의 계약이 완료된 경우 CA는 서버가 제출한 데이터를 검증하고 CA의 개인키로 서버에서 제출한 정보를 암호화해서 인증서를 만들어 제공한다. 서버는 인증서를 가지게 된다.
3. CA는 클라이언트에게 자신의 공개키를 제공한다.
4. 사용자가 서버에 접속하면 서버는 자신의 인증서를 클라이언트에게 보낸다. 예를 들어, 클라이언트가 index.html 파일을 요청했다면, 서버의 정보를 CA의 개인키로 암호화한 인증서(2번에서 받은거)를 받게 되는 것이다.
5. 클라이언트는 3번에서 미리 알고 있던 인증기관의 공개키로 인증서를 해독하여 검증한다. 그러면 서버의 정보와 서버의 공개키를 알 수 있게 된다.
6. 이렇게 얻은 서버의 공개키로 클라이언트는 대칭키를 만들고 암호화해 다시 서버에 보낸다.
7. 서버는 서버의 개인키로 암호문을 해독하여 대칭키를 얻게 되고, 이제 대칭키로 클라이언트와 데이터를 주고받는다.
8. 세션이 종료되면 대칭키를 폐기시킨다.

위에서 공부한 공개키, 개인키, 대칭키를 잘 생각하면서 조금만 더 간단하게 해보자.

1. 서버가 공개키와 개인키 생성 → CA와 계약
2. CA는 서버의 데이터를 검증하고 CA만의 개인키로 해당 정보를 암호화해서 인증서로 만들고 서버에게 인증서 전달

   → 즉, 인증서 = 암호화된 서버 정보

   → CA의 개인키로 암호화 했으므로 CA의 공개키로 복호화 가능
3. CA는 클라이언트에게 CA의 공개키 제공
4. 사용자가 서버 접속 시, 서버는 인증서를 클라이언트에게 제공
5. 클라이언트는 CA의 공개키로 인증서 복호화 → 서버의 정보, 서버의 공개키 알 수 있음
6. 클라이언트는 서버의 공개키로 대칭키를 만들고 이를 암호화해 서버에 전달

   → 서버의 공개키로 암호화 했으므로 서버의 개인키로 복호화 가능
7. 서버는 서버의 개인키로 암호화된 대칭키를 해독해 대칭키를 얻고, 이제 대칭키를 이용해 데이터를 주고받음