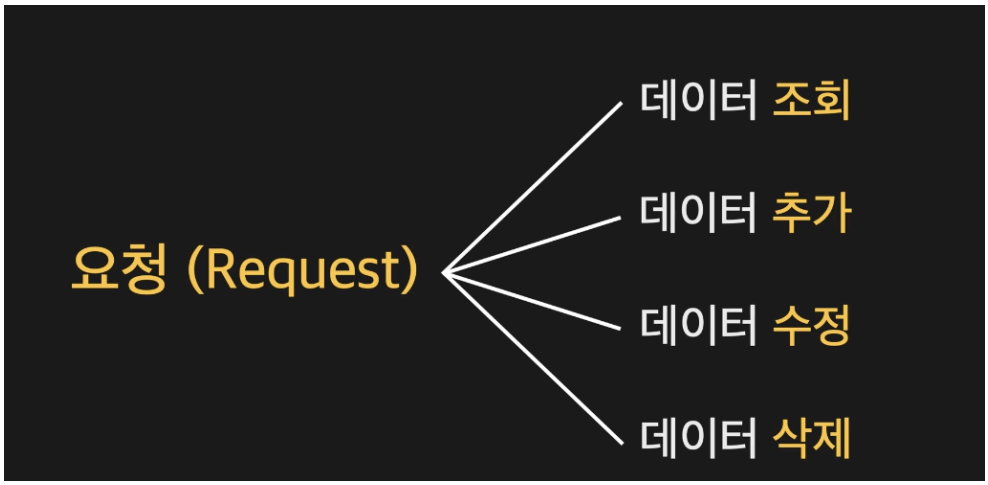
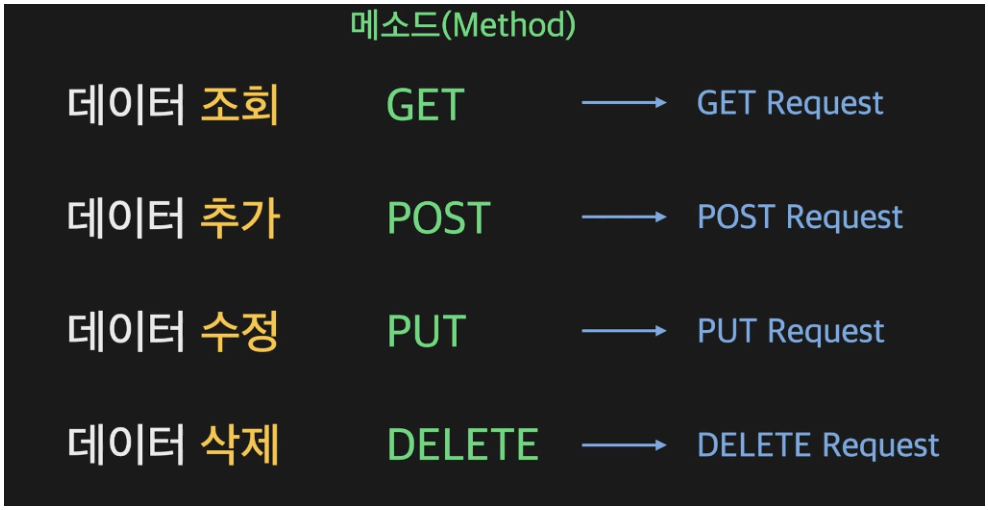
Request



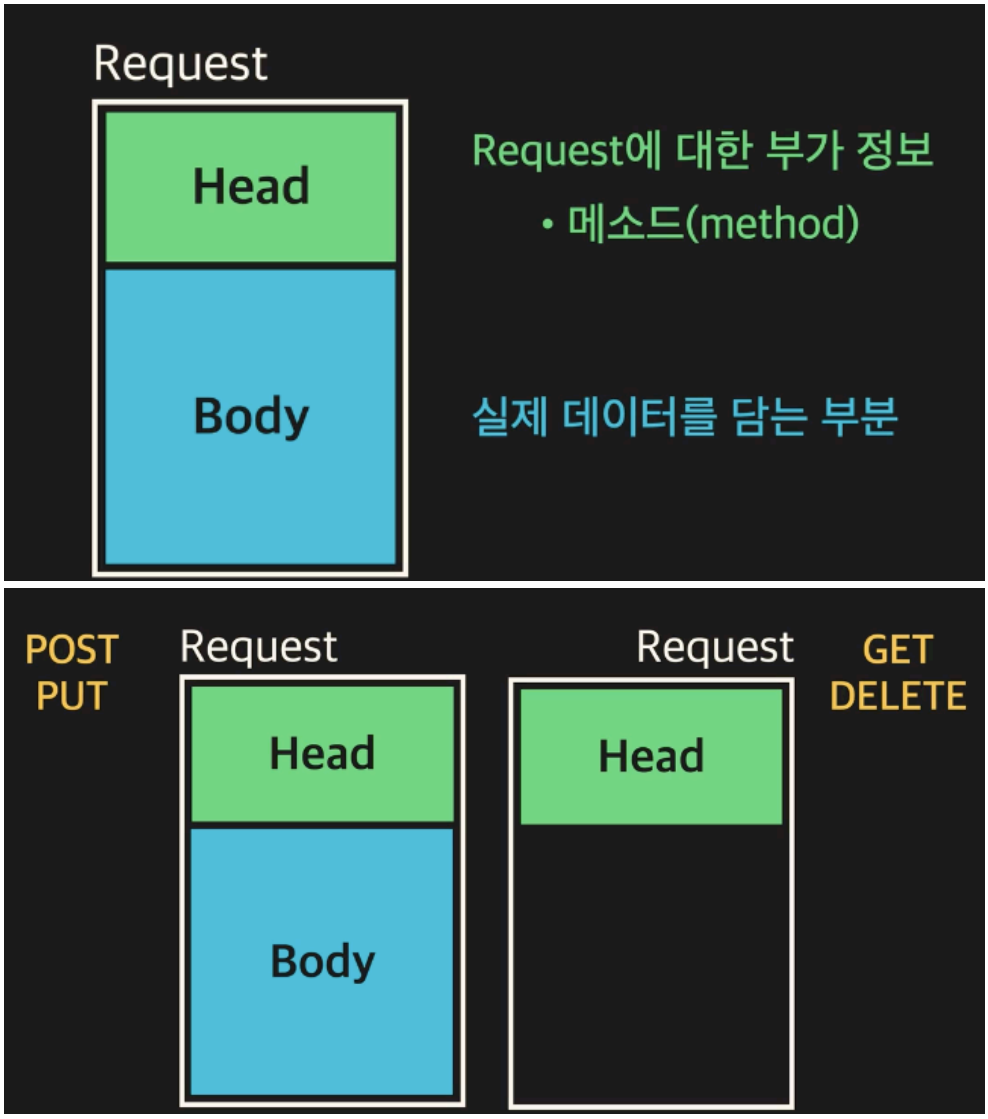


결국 각각의 메소드에 대해서 서버는 보통 그에 맞는 데이터 관련 조작을 하겠죠? 만약 서버가 데이터베이스를 사용한다면 CRUD 작업을 하게 될 겁니다.

CRUD란 Create-Read-Update-Delete의 약자로 데이터베이스 관점에서 데이터에 관한 처리를 나타낸 합성어인데요.

각 메소드는 각 데이터 관련 작업에 이렇게 대응됩니다.

Request



기존 데이터를 조회하는 리퀘스트 - GET

새 데이터를 추가하는 리퀘스트 - POST

기존 데이터를 수정하는 리퀘스트 - PUT

기존 데이터를 삭제하는 리퀘스트 - DELETE

status code

모든 상태 코드(Status Code)는 각각 그에 대응되는 상태 메시지(Status Message)를 갖고 있다.

상태 코드는 100번대부터 500번대까지 존재합니다.

Request

100번대

서버가 클라이언트에게 정보성 응답(Informational response)을 줄 때 사용되는 상태 코드들입니다.

100 Continue : 클라이언트가 서버에게 계속 리퀘스트를 보내도 괜찮은지 물어봤을 때

- 클라이언트가 용량이 좀 큰 파일을 리퀘스트의 바디에 담아 업로드하려고 할 때
- 서버에게 미리 괜찮은지를 물어보는 경우가 있다고 할 때,
- 서버가 이 100번 상태 코드의 리스폰스를 주면 그제서야 본격적인 파일 업로드를 시작합니다.

101 Switching Protocols : 클라이언트가 프로토콜을 바꾸자는 리퀘스트를 보냈을 때, 서버가 '그래요, 그 프로토콜로 전환하겠습니다'라는 뜻을 나타낼 때 쓰이는 상태 코드입니다.

200번대

클라이언트의 리퀘스트가 성공 처리되었음을 의미하는 상태 코드

200 OK : 리퀘스트가 성공적으로 처리되었음을 포괄적으로 의미하는 상태 코드

201 Created : 리퀘스트의 내용대로 리소스가 잘 생성되었다는 뜻

- POST 리퀘스트가 성공한 경우에 200번 대신 201번이 올 수도 있다.

202 Accepted : 리퀘스트의 내용이 일단은 잘 접수되었다는 뜻입니다. 즉, 당장 리퀘스트의 내용이 처리된 것은 아니지만 언젠가 처리할 것이라는 뜻인데요. 리퀘스트를 어느 정도 모아서 한번에 실행하는 서버인 경우 등에 이런 응답을 줄 수도 있다.

300번대

클라이언트의 리퀘스트가 아직 처리되지 않았고, 리퀘스트 처리를 원하면 클라이언트 측의 추가적인 작업이 필요함을 의미하는 상태 코드

301 Moved Permanently : 리소스의 위치가 바뀌었음을 나타낸다. 보통 이런 상태 코드가 있는 리스폰스의 헤드에는 Location이라는 헤더도 일반적으로 함께 포함되어 있다. 그리고 그 헤더의 값으로 리소스에 접근할 수 있는 새 URL이 담겨있는데요. 대부분의 브라우저는 만약 GET 리퀘스트를 보냈는데 이런 상태 코드가 담긴 리스폰스를 받게 되면, 헤드에 포함된 Location 헤더의 값을 읽고, 자동으로 그 새 URL에 다시 리퀘스트를 보내는 동작(리다이렉션, redirection)을 수행합니다.

302 Found : 리소스의 위치가 일시적으로 바뀌었음을 나타낸다. 이 말은 지금 당장은 아니지만 나중에는 현재 요청한 URL이 정상적으로 인식될 것이라는 뜻입니다.

304 Not Modified : 브라우저들은 보통 한번 리스폰스로 받았던 이미지 같은 리소스들을 그대로 내부에 저장하고 있다. 그리고 서버는 해당 리소스가 바뀌지 않았다면, 리스폰스에 그 리소스를 보내지 않고 304번 상태 코드만 헤드에 담아서 보냄으로써 '네트워크 비용'을 절약하고 브라우저가 저장된 리소스를 재활용하도록 하는데, 이 상태 코드는 웹에서 '캐시

(cache)'라는 주제에 대해서 공부해야 정확하게 이해할 수 있습니다. 당장 배울 내용은 아니니까 넘어갈게요. 혹시 관심이 있는 분들은 이 링크를 참조하세요.

400번대

리퀘스트를 보내는 클라이언트 쪽에 문제가 있음을 의미하는 상태 코드

400 Bad Request : 말그대로 리퀘스트에 문제가 있음을 나타냅니다. 리퀘스트 내부 내용의 문법에 오류가 존재하는 등의 이유로 인해 발생한다.

401 Unauthorized : 아직 신원이 확인되지 않은(unauthenticated) 사용자로부터 온 리퀘스트를 처리할 수 없다는 뜻한다.

403 Forbidden : 사용자의 신원은 확인되었지만 해당 리소스에 대한 접근 권한이 없는 사용자라서 리퀘스트를 처리할 수 없다는 뜻

404 Not Found : 해당 URL이 나타내는 리소스를 찾을 수 없다는 뜻
보통 이런 상태 코드가 담긴 리스폰스는 그 바디에 관련 웹 페이지를 이루는 코드를 포함하고 있는 경우가 많습니다.
예를 들어, 다음과 같이 https://www.google.com/abc와 같이 존재하지 않는 URL에 접속하려고 하면 이런 페이지가 보이는 것을 알 수 있습니다.

405 Method Not Allowed : 해당 리소스에 대해서 요구한 처리는 허용되지 않는다는 뜻. 만약 어떤 서버의 이미지 파일을 누구나 조회할 수는 있지만 아무나 삭제할 수는 없다면 GET 리퀘스트는 허용되지만, DELETE 메소드는 허용되지 않는 상황인 건데요. 그런데 만약 그 이미지에 대한 DELETE 리퀘스트를 보낸다면 이런 상태 코드를 보게될 수도 있다.

413 Payload Too Large : 현재 리퀘스트의 바디에 들어있는 데이터의 용량이 지나치게 커서 서버가 거부한다는 뜻

429 Too Many Requests : 일정 시간 동안 클라이언트가 지나치게 많은 리퀘스트를 보냈다는 뜻. 서버는 수많은 클라이언트들의 리퀘스트를 정상적으로 처리해야 하기 때문에 특정 클라이언트에게만 특혜를 줄 수는없더, 따라서 지나치게 리퀘스트를 많이 보내는 클라이언트에게는 이런 상태 코드를 담은 리스폰스를 보낼 수도 있다.

500번대

서버 쪽의 문제로 인해 리퀘스트를 정상적으로 처리할 수 없음을 의미하는 상태 코드

500 Internal Server Error : 현재 알 수 없는 서버 내의 에러로 인해 리퀘스트를 처리할 수 없다는 뜻

503 Service Unavailable : 현재 서버 점검 중이거나, 트래픽 폭주 등으로 인해 서비스를 제공할 수 없다는 뜻

Content-Type 헤더

Content-Type 헤더는 현재 리퀘스트 또는 리스폰스의 바디에 들어 있는 데이터가 어떤 타입인지를 나타낸다.

Content-Type 헤더의 값은 '주 타입(main type)/서브 타입(sub type)'의 형식으로 나타나는데, 우리가 흔히 만나게 될 Content-Type 헤더의 값으로는 다음과 같다.

Request

주 타입이 text인 경우(텍스트)

- 일반 텍스트 : text/plain
- CSS 코드 : text/css
- HTML 코드 : text/html
- JavaScript 코드 : text/javascript ...
- 주 타입이 image인 경우(이미지)
- image/bmp : bmp 이미지
- image/gif : gif 이미지
- image/png : png 이미지 ...
- 주 타입이 audio인 경우(오디오)
- audio/mp4 : mp4 오디오
- audio/ogg : ogg 오디오 ...
- 주 타입이 video인 경우(비디오)
- video/mp4 : mp4 비디오
- video/H264 : H264 비디오 ...

위 타입들에 속하지 않는 것들은, 보통 application이라고 하는 주 타입으로

주 타입이 application인 경우

- application/json : JSON 데이터
- application/octet-stream : 확인되지 않은 바이너리 파일

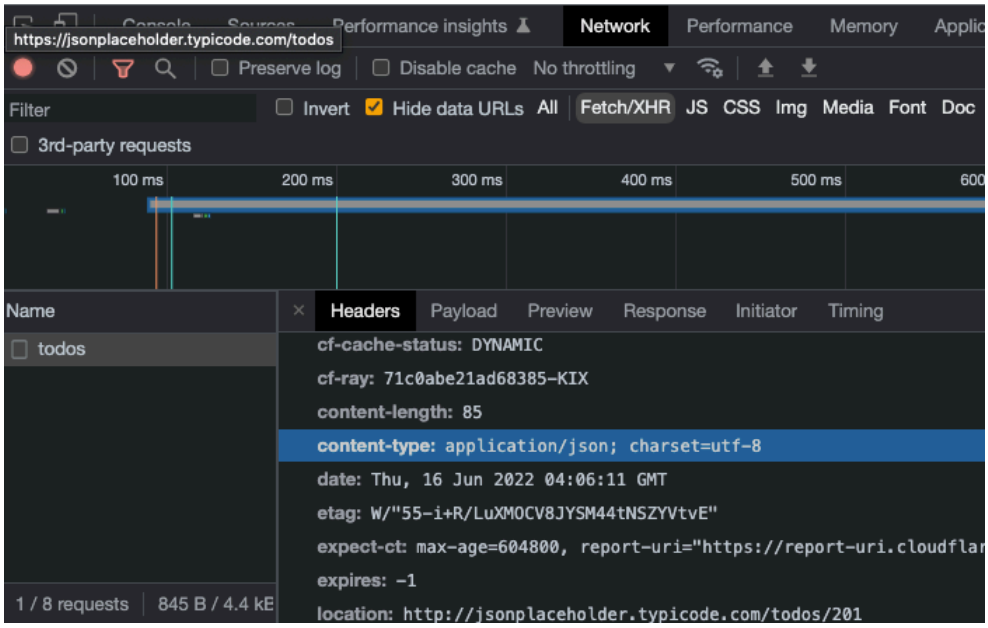
Content-Type 설정해보기

```
const member = {
  "userId": 1,
  "title": "delectus aut autem",
  "completed": false,
};

fetch('https://jsonplaceholder.typicode.com/todos', {
  method: 'POST',
  body: JSON.stringify(member),
})
  .then((response) => response.text())
  .then((result) => { console.log(result); });
```

```
const member = {
  "userId": 1,
  "title": "delectus aut autem",
  "completed": false,
};

fetch('https://jsonplaceholder.typicode.com/todos', {
  method: 'POST',
```

Request

```
  headers: { // 추가된 부분
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(member),
})
  .then((response) => response.text())
  .then((result) => { console.log(result); });
```



비동기 실행

동기실행

한번 시작한 작업은 다 처리하고 나서야, 다음 코드로 넘어가는 일반적인 방식 실행을 '동기 실행'이다.

동기 실행은 한번 시작한 작업을 완료하기 전까지 코드의 실행 흐름이 절대 그 다음 코드로 넘어가지 않는다. 일단 시작한 작업을 완벽하게 처리하고 난 다음에야 그 다음 코드로 실행 흐름이 넘어간다. 따라서 동기 실행의 경우 코드가 보이는 순서대로 실행된다.

비동기실행

비동기 실행은 한번 작업을 시작해두고, 그 작업이 완료되기 전이더라도 콜백만 등록해두고, 코드의 실행 흐름이 바로 그 다음 코드로 넘어간다. 그리고 추후에 특정 조건이 만족되면 콜백이 실행됨으로써 해당 작업을 완료하는 방식이다. 따라서 비동기 실행에서는 코드가 꼭 등장하는 순서대로 실행되는 것이 아니다. 그래서 코드를 해석할 때 주의해야 하는데요.

비동기 실행이 진행 되는 매서드

- fetch() 요청의 시작점 .then() : 이어서 들어가는 작업
- setTimeout 함수
- setInterval 함수
- addEventListener()

fetch('주소', 명령)

get (조회) / post (추가)
/ put (수정) / delete (삭제)

post와 put은 body 부분이 필요함

Promise객체

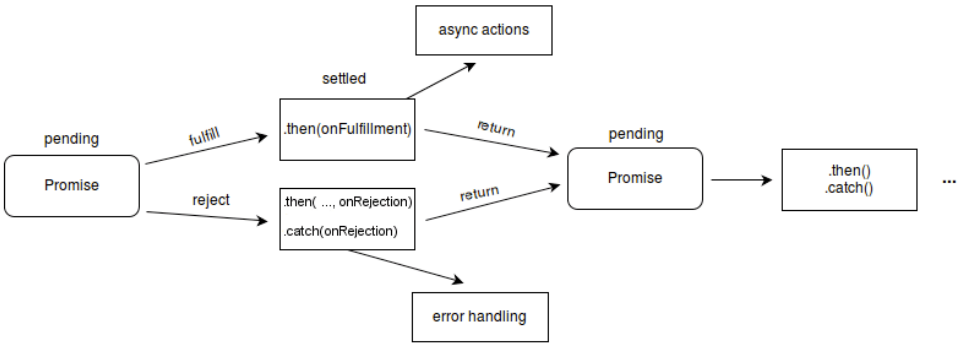
생성된 시점에는 알려지지 않았을 수도 있는 값을 위한 대리자로, 비동기 연산이 종료된 이후에 결과 값과 실패 사유를 처리하기 위한 처리기를 연결할 수 있습니다. 프로미스를 사용하면 비동기 메서드에서 마치 동기 메서드처럼 값을 반환할 수 있습니다. 다만 최종 결과를 반환하는 것이 아니고, 미래의 어떤 시점에 결과를 제공하겠다는 '약속'(프로미스)을 반환합니다.

fetch('주소', {method:'명령어', body: 데이터}

대기(pending): 이행하지도, 거부하지도 않은 초기 상태.

이행(fulfilled): 연산이 성공적으로 완료됨.

거부(rejected): 연산이 실패함.



Promise.prototype.catch()

에러 상황에 실행 되는 것

마지막에 위치 -> 위에서 발생하는 모든 에러를 처리할 수 있도록

프로미스에 거부 처리기 콜백을 추가하고, 콜백이 호출될 경우 그 반환값으로 이행하며 호출되지 않을 경우, 즉 이전 프로미스가 이행하는 경우 이행한 값을 그대로 사용해 이행하는 새로운 프로미스를 반환합니다.

Promise.prototype.then()

프로미스에 이행과 거부 처리기 콜백을 추가하고, 콜백이 호출될 경우 그 반환값으로 이행하며 호출되지 않을 경우 (onFulfilled, onRejected 중 상태에 맞는 콜백이 함수가 아닐 경우) 처리된 값과 상태 그대로 처리되는 새로운 프로미스를 반환합니다.

Promise.prototype.finally()

끝

프로미스의 이행과 거부 여부에 상관없이 처리될 경우 항상 호출되는 처리기 콜백을 추가하고, 이행한 값 그대로 이행하는 새로운 프로미스를 반환합니다.

async function

async function 선언은 AsyncFunction객체를 반환하는 하나의 비동기 함수를 정의한다. 비동기 함수는 이벤트 루프를 통해 비동기적으로 작동하는 함수로, 암시적으로 Promise를 사용하여 결과를 반환한다 그러나 비동기 함수를 사용하는 코드의 구문과 구조는, 표준 동기 함수를 사용하는것과 많이 비슷하다.

```
async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
  // expected output: "resolved"
}

asyncCall();
```

throw문

사용자 정의 예외를 발생(throw)할 수 있다. 예외가 발생하면 현재 함수의 실행이 중지되고 (throw 이후의 명령문은 실행되지 않는다), 제어 흐름은 콜스택의 첫 번째 catch 블록으로 전달되며,호출자 함수 사이에 catch 블록이 없으면 프로그램이 종료한다.