Javascript Fetch API

이한울
Jan 21, 2018 · 14 min read

개요

그 동안 WEB에서 어떤 리소스를 비동
Request)객체를 사용했어야 했었는데,

XHR은 잘 디자인되어 있는 API가 아니다. 요청의 상태나 변경을 구독하려면 Event를 등록해서 변경사항을 받아야 했고

요청의 성공, 실패 여부나 상태에 따라 처리하는 로직이 들어가기 좋지 않았다.

이를 보완하기 위해서 Fetch API를 도입하였는데 이는 HTTP 요청에 최적화 되어 있고 상태도 잘 추상화 되어 있고

Promise를 기반으로 되어 있기때문에 상태에 따른 로직을 추가하고 처리하는데에 최적화 되어 있다.

Baisc Use

```
fetch('http://hanur.me/users')
.then(res => res.json())
.then(data => data.filter(item => item.isRequired));
```

기본적으로 위와 같이 사용할 수 있다.

Basic

Fetch API는 3개의 Interface를 도입하고 있는데 Headers, Request, Response 이다.

이는 곧 HTTP의 개념과 대응되는 Interface이다.

Headers

Headers 객체는 HTTP Header와 대응되는 객체이다.

```
const reqHeader = new Header
const content = "HI";
reqHeader.append("Content-Ty
reqHeader.append("Content-Le
```


위 처럼 Headers 객체를 만들고 apper
보를 추가할 수 있다.

혹은 Append가 아닌 생성자로 Object

```
const content = "HI";
const reqHeader = new Headers({
  "Content-Type": "application/json",
  "Content-Length": content.length.toString(),
});
```

더 자세히 보기위해서 Headers 객체의 Prototype을 보자

```
append: function append()
delete: function delete()
entries: function entries()
forEach: function forEach()
get: function get()
getAll: function getAll()
has: function has()
keys: function keys()
set: function set()
values: function values()
constructor: function Headers()
Symbol(Symbol.iterator): function ()
Symbol(Symbol.toStringTag): "Headers"
```

가장 눈에 띄는것은 Symbol.iterator를 구현한 Iterable 객체라는 점이다.

아마 기본적으로 key, value로 Header를 Set하고 있으니 해당 아이템을 List로 Iterator를 돌릴 수 있게 제공하는것 같다.

```
const reqHeader = new Headers({
  "Content-Type": "application/json",
  "Content-Length": "30",
});
```

```
for(const item of reqHeader){
  console.log(item)
}

// console
// -> ["content-type", "appl
// -> ["content-length", "30
```


for of문은 Symbol.iterator를 구현하고 item은 내 예상과 다르게 Array에 Key,

내가 예상해던건 Object에 {key: value 뽑는게 쉽지 않아서 Array에 담은것 같다. (Object.keys나 Object.values를 이용하면 되긴하지만 일을 2번하는 느낌이므로...)

entries()

해당 메서드는 Iterator를 리턴해주는데 `Headers[Symbol.iterator]()` 와 같은 결과를 반환한다.

forEach()

forEach을 따로 제공해주는데 Iterator를 돌리는것과 또 다르다.

```
const reqHeader = new Headers({
  "Content-Type": "application/json",
  "Content-Length": "30",
});

reqHeader.forEach((value, key, header) => console.log(value, key,
header));
```

callback 함수의 파라미터로 value, key, header 순으로 들어오게 된다. header는 현재 forEach를 돌고있는 해당 header 객체이다

keys()

```
const reqHeader = new Headers({
  "Content-Type": "application/json",
  "Content-Length": "30",
});

const keys = reqHeader.keys();
```

```
for(const key of keys) {
  console.log(key)
}
```


keys()는 Header에 등록된 Request H
드이다. 만약 중복된 키로 여러 개의 V
하나의 key만 반환한다.

```
const reqHeader = new Header
  "Content-Type": "applicati
  "Content-Type": "plain/tex
  "Content-Length": "30",
});

const keys = reqHeader.keys();

for(const key of keys) {
  console.log(key)
}

// console
// -> "content-type"
// -> "content-length"
```

values()

values()는 Header에 등록된 Request Header value를 반환하는 Iterator를 반환하는 메서드이다.

만약 중복된 key로 여러 value를 등록했을 경우 하나의 String에 ,Seperator로 이어서 반환된다.

```
const reqHeader = new Headers({
  "Content-Type": "application/json",
  "Content-Type": "plain/text",
  "Content-Length": "30",
});

const values = reqHeader.values();

for(const value of values) {
  console.log(value)
}

// console
// -> "application/json,plain/text"
```

```
// -> "30"
```


먼저 등록된 value가 앞에 나오고 나중

set()과 append()의 차이점

set()과 append()가 겉으로 보기에는

두개의 API가 기본적인 사용법은 같은

```
const reqHeader = new Headers();
reqHeader.append("Content-Type", "application/json");
reqHeader.append("Content-Type", "hihi");
reqHeader.append("Content-Length", "30");

for(const value of reqHeader.values()){
  console.log(value);
}

// console
// -> "application/json,hihi"
// -> "30"

reqHeader.set('Content-Type', 'good');
for(const value of reqHeader.values()){
  console.log(value);
}
// console
// -> "good"
// -> "30"
```

위 Code를 보면 알 수 있는 것처럼 append()는 설정을 추가해주는 메서드이고 set()은 해당 key를 이용해서 set해주고

만약 같은 key가 있다면 덮어써주는 메서드이다.

guard

Headers 객체는 우리가 Request를 보낼때도 사용하고 Response로 받는 경우에도 사용하게 됩니다.

우리가 직접 만든 Headers는 수정이 가능하지만, 만약 Response로 받은 Header의 경우 우리가 임의로 수정 할 수 없는게 맞습니다.

Server에서 내려준 Response에 대한 Header 이기때문에 그대로 사용하는게 맞습니다. 그러므로 Header에는 Header를 수정

여부를 판단하는 guard가 존재합니다
아서 실제로 guard 값을 조회할 방법

하지만 내부적으로 존재하는 값입니

- none: 기본값.
- request: Request에 존재하는 Hea
- request-no-cors: no-cors mode로 생성된 Request의 Headers 객체를 위한 값.
- response: Response 객체에 있는 Headers 객체를 위한 값.
- immutable: ServiceWork를 위해서 사용되는 값. Header가 Read-only라는 것을 나타낸다.

가장 기본값은 “none”이다. 이는 모든 변경을 사실상 허용한다는 의미이다.

“request”는 request를 위한 Header name이 아닌 경우 변경을 금지한다.

예를 들어서 request의 Headers의 guard 속성은 “request”이다.

그래서 Request Header에 정의되는 속성이 아닌 Response Header 값을 세팅한다거나 하면 Type Error를 발생시킨다.

이런 방식으로 guard가 작동하는데 확인할 수 없다는 점이 아쉬운 점이다.

Request

Request는 HTTP 요청을 통해 자원을 가져오는 인터페이스이다.

Request는 URL, Header, body가 필요하다. 그리고 Request에 대한 mode 제한과 certificate 관련 설정도 추가할 수 있다.

```
const req = new Request("/api/posts", {
  method: "GET",
  headers: new Headers({
    "content-type": "application/json",
  }),
  body: {
    name: "LeeHanur",
```

```
    }
});
```


이런 느낌으로 사용할 수 있다. 물론 R
가는건 아니다. Request 객체는 어떠
제로 Fetching 하는건 fetch Method를

```
const req = new Request("/ap
  method: "GET",
  headers: new Headers({
    "content-type": "applica
  }),
  body: {
    name: "LeeHanur",
  }
});
// is not Fetched

fetch(req).then(res => res.json()).then(data =>
console.log(data));
// is fetched Data!
```

이런 느낌이라고 생각하면 된다.

Request 객체의 첫번째 인자는 호출한 Path가 들어가고, 두번째 인자에는 Request에 대한 정보가 들어가는데,

- method
- headers
- body
- mode
- cache
- credentials
- redirect
- referrer
- integrity

method

method는 HTTP Method와 동일한 스
PUT / DELETE / OPTION / PATCH 등

만약 지정을 안할 경우 기본값은 "GET
대문자로 Uppercase된다.


```
const req = new Request('/ap
  method: 'get'
});

req.method // "GET"
```

headers

headers는 Request Header를 지정해주는 곳인데 2가지 형식으로 넣을 수 있다. Object literal과 Headers 객체의 인스턴스를 넣을 수 있다.

```
const req = new Request('/api/posts', {
  method: "GET",
  headers: {
    'content-type': 'application/json',
  }
});

const req2 = new Request('/api/posts', {
  method: 'GET',
  headers: new Headers({
   'content-type': 'application/json',
  })
})
```

body

body는 HTTP Request에 실을 데이터인데 여러가지 타입을 넣을 수 있다. 스팩을 보면 아래와 같이 명시되어 있다.

```
typedef (Blob or BufferSource or FormData or URLSearchParams or
ReadableStream or USVString)
```

Blob도 보낼 수 있는 부분을 보면 파일전송까지 모두 고려한것으로 볼 수 있고, FormData와 URLSearchParams도 넣을

mode

mode는 Request시 어떤 Origin에 있는 분이다.

- same-origin
- no-cors(Default)
- cors

same-origin은 현재 Origin과 같은 Origin에만 요청 할 수 있는 설정이다. 이를 이용하면 해당 Request는 같은 Origin에만 동작한다는걸 보장할 수 있다.

no-cors는 기본설정으로, 일반적으로 CDN에 있는 데이터를 가져온다거나 이미지를 불러온다거나 하는 기본적인 동작들을 할 수 있다. GET, POST, HEAD 이외에 다른 Method는 지원하지 않는다.

cors는 여러 다른 API에 접근할때 사용할 mode이다. Response Header가 일부 공개되지 않을 수 있지만 Body는 모두 접근가능하도록 Response가 온다.

Response

Response는 fetch를 호출하면 가져올 수 있는 객체인데, Response의 경우는 ServiceWorker가 아니면 생성해서 쓰는것이 크게 의미가 없다.

- status
- statusText
- ok
- headers
- type

status

status는 HTTP Response Code를 담고 있다. 일반적으로 성공적이였다면 200이 떨어지게 될것이다.

statusText

statusText는 기본값은 “ok”이고 상황

ok

ok는 Status의 200299의 값을 추상화
가지게 된다.

headers

Response Headers 이기 때문에 header

type

type은 Response 객체의 type을 말한다.

- basic
- cors
- error
- opaque

basic은 가장 기본적인 속성이고 error가 난 Response가 아니고, Request시 mode의 값이 cors가 아니라면 basic이 오게 될 것이다. 가능한 모든 Headers 속성에 접근할 수 있다.

cors는 Request의 mode가 cors일때 나오는 값인데, 이 때는 Headers의 일부 값에 대한 접근이 제한된다.

error는 Response.error()를 호출했을때 나오는 type이다.

opaque는 Request mode를 no-cors로 설정했을때 가지게 되는 type이다.

Body를 다루기

Request와 Response 모두 Body값이 존재하는데 Body는 아래와 같은 종류로 설정할 수 있다.

- ArrayBuffer
- ArrayBufferView (Uint8Array와 친구들)
- Blob/File

- string
- URLSearchParams
- FormData

이런 다양한 종류의 Body를 다루기위

- arrayBuffer()
- blob()
- json()
- text()
- formData()

JavaScript Fetch API