# Fetch API

prayme · 2020년 7월 11일



Fetch JavaScript ajax



# Fetch API 이제 그만 좀 찾아보자

웹 개발을 할 때 ajax 통신을 자주 사용한다. ajax를 사용할 때 XHR, JQuery, Fetch 등의 선택지가 있지만 셋 다 사용해봤을 때 JQuery와 Fetch가 당연한 소리지만 압도적으로 좋은 것 같다. (생산성 측면에서) 앞으로 추세가 JQuery를 쓰지 말자는 추세이기 때문에 Fetch를 사용해야겠다. 그런데 Fetch를 쓸 때 마다 검색을 통해 사용법을 다시 숙지해야했다. 문법을 찾아보는 것이 아니라 동작 방식을 다시 살펴봐야했다. 이번 기회에 Promise, async, await, fetch에 대해서 기록하고 다음 부터는 문법만 검색하자

# MDN 문서 읽어보기

Fetch는 네트워크 요청/응답에 관련된 일반적인 Request/Response Object를 제공한다. 이 말은 범용성이 좋다는 말이다. 캐시, 웹 서비스 핸들링 등 Response를 프로그래밍 언어로 조작하는 모든 것을 허용한다는 뜻이다. 또한 CORS, HTTP Origin header semantics와 같은 개념들도 정의해놨고 이것들을 따로 수정할 수도 있다.

`WindowOrWorkkerGlobalScope.fetch()` 메소드를 사용하면 요청과 응답을 만들 수 있다. (`WindowOrWorkerGlobalScope` 는 생략되는 듯 하다) 이 메소드는 여러 인터페이스에서 구현이

되어 있는데 특히 `Window`, `WorkerGlobalScope` 에서 구현이 되어 있다. `fetch()` 로 거의 모든 상황에서 ajax 통신을 가능하게 한다.

`fetch()` 는 2개의 매개변수를 받는데 첫번째는 URL이고 두번째는 Option이다. URL은 필수 매개변수이다. 그리고 이 녀석은 ajax 통신이 성공하든 실패하든 `Response` 로 분해할 수 있는 `Promise` 를 리턴한다.

# JQuery와 가장 다른점 세가지

- `fetch()` 가 반환하는 `Promise` 는 response가 HTTP 404, 500 같은 HTTP error status여도 거부하지 않고 다 받아온다.
- cross-site cookies를 받지 않는다. `fetch()` 는 cross-site session을 설정할 수 없다. 다른 사이트의 Set-Cookie 헤더는 자동으로 무시한다.
- credential init 옵션을 설정하지 않으면 cookie를 전송하지 않는다.

Fetch에는 `fetch()`, `Headers`, `Request`, `Response` 인터페이스가 존재한다.

# Fetch Interfaces

fetch에는 4가지의 인터페이스와 1가지의 메소드밖에 없다. 나열하고 하나씩 살펴보자

- Body
- Headers
- Requests
- Response
- fetch()

## Body

mixin 타입이라고 한다. Request, Response 두개 모두에서 사용된다. 자세한건 문서참고.. 딱 가지의 속성을 가지고 있는데 **`Body.body`** , **`Body.bodyUsed`** 이다. Body에는 5가지의 메소드가 존재한다.

- **`Body.arrayBuffer()`**
- **`Body.blob()`**
- **`Body.formData()`**
- **`Body.json()`**
- **`Body.text()`**

## Headers

Request와 Response의 Headers instance를 생성할 수 있다. `let myHeaders = new Headers();` 같이 생성할 수 있다. 생성한 Header에 속성들을 추가, 제거, 조회 할 수 있다. 다음과 같은 메소드를 제공한다. `append()`, `delete()`, `entries()`, `forEach()`, `get()`, `has()`, `keys()`, `set()`, `values()`, `getAll()`

### 예제

```
var myHeaders = new Headers();

myHeaders.append('Content-Type', 'text/xml');
myHeaders.get('Content-Type') // should return 'text/xml'

var myHeaders = new Headers({
    'Content-Type': 'text/xml'
});

// or, using an array of arrays:
myHeaders = new Headers([
    ['Content-Type', 'text/xml']
]);

myHeaders.get('Content-Type') // should return 'text/xml'
```

## Request

Request instance는 요청 headers의 properties를 포함한다. 자세한건 문서참고

### 예제

```
const request = new Request('https://www.mozilla.org/favicon.ico');

const URL = request.url;
const method = request.method;
const credentials = request.credentials;

fetch(request)
  .then(response => response.blob())
  .then(blob => {
    image.src = URL.createObjectURL(blob);
  });
```

## Response

fetch()는 Promise를 리턴하는데 Promise에서 값을 추출하면 Response를 얻을 수 있다.
마찬가지로 자세한 사항은 문서를 확인하자

# fetch() 사용해보자

## GET 요청하기

간단한 express 서버와 프론트의 ajax 통신을 살펴보자

### express route 코드

```
/* GET home page. */
router.get('/', function (req, res, next) {
  res.render('index', { title: 'Hello, Fetch!' });
});

router.get('/title/:title', (req, res, next) => {
  res.json({ title: req.params.title });
})
```

### index 페이지

```
<h1 id="title">{{{title}}}</h1>
  <input id="input" type="text">
  <input type="button" value="submit" onclick="submit()">

  <script>
    let submit = () => {
      let input = document.getElementById('input');
      let target = document.getElementById('title');
      fetch('/title/' + input.value)
        .then(res => res.json())
        .then(json => target.innerHTML = json.title);

    }
  </script>
```

- 처음 입장 했을 때 다음과 같이 Hello, Fetch!로 반겨준다.


그리고 submit 버튼을 클릭하면 왼쪽에 입력값을 파라미터로 GET 통신을 진행한다. 통신에 성공하면 서버에서 body에 JSON을 담아서 반환하고 fetch는 promise를 통해 json의 값을 뽑아낸다.

```
fetch('/title/'+title)
        .then(res => res.json())
    .then(json => target.innerHTML = json.title);
```

아까 언급 했듯이 `fetch()` 는 Promise를 리턴한다. 첫번째 then에서 Response의 json을 return하고 다음 then에서 json의 값을 뽑아내서 Hello, Fetch!에 해당하는 값을 변경한다.


# POST 요청하기

- express route 코드에 다음 코드를 추가하자

```
router.post('/title', (req, res, next) => {
  res.json({ title: req.body.title});
})
```

- 프론트의 코드도 다음과 같이 수정하자

```
let submit = () => {
      let input = document.getElementById('input');
      let target = document.getElementById('title');
      fetch('/title', {
        method: 'post',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ 'title': input.value }),
      })
        .then(res => res.json())
        .then(json => target.innerHTML = json.title)
        .catch(error => console.error('Error: ', error));
    }
```

- 전송 시 화면


# fetch 에러처리와 비동기 요청

이 포스팅처럼 fetch안에서 fetch를 또 호출하는 등 코드가 길어지고 복잡해지면 디버깅하기 힘들어진다고 한다. 필자의 수준에서는 아직 느껴본 적이 없는 감정이지만.. 어쨌든 그런 상황을 해결하기 위해 ES7에서부터 도입된 async/await를 활용해서 fetch API를 사용하고 ajax를 사용해보자

# async와 await가 뭐지?

일단 다음 코드를 살펴보자

```
let submit = async () => {
      let input = document.getElementById('input');
      let target = document.getElementById('title');
      let replaceTitle = json => target.innerHTML = json.title;
      let url = '/title'
      let options = {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({'title': input.value}),
      }

      try {
        let response = await fetch(url, options);
        let json = await response.json();
        let task = await replaceTitle(json);
      } catch (err) {
        console.log(err);
      }
    }
```

POST 요청하는 코드를 수정해봤다. fetch는 기본적으로 비동기로 동작하기 때문에 가끔 원하는대로 순서대로 동작하지 않을 때가 있다. 그럴 때 순서를 보장받기 위해서 async/await를 사용한다.
await는 async 선언이 된 함수 안에서만 사용이 가능하며 Promise값을 기다렸다가 Promise값에서 결과값을 추출해준다. 그리고 async를 선언한 함수는 반드시 Promise를 리턴한다. 이렇게 간단하게 순서를 보장받을 수 있기 때문에 디버깅, 예외처리가 용이해진다.

**황성찬**

잘하고 싶은 사람

다음 포스트
**conda란?**

## 0개의 댓글

댓글을 작성하세요

댓글 작성