30일 8월 2020

fetch

자바스크립트를 사용하면 필요할 때 서버에 네트워크 요청을 보내고 새로운 정보를 받아오는 일을 할 수 있습니다.

네트워크 요청은 다음과 같은 경우에 이뤄집니다.

- 주문 전송
- 사용자 정보 읽기
- 서버에서 최신 변경분 가져오기
- 등등

그런데 이 모든 것들은 페이지 새로 고침 없이도 가능합니다.

AJAX(**A**synchronous **J**avaScript **A**nd **X**ML, 비동기적 JavaScript와 XML)라는 용어를 들어보신 분이 있으실 겁니다. AJAX는 서버에서 추가 정보를 비동기적으로 가져올 수 있게 해주는 포괄적인 기술을 나타내는 용어로, 만들어진 지 오래되었습니다. AJAX에 XML이 포함된 이유가 바로 이 때문이죠.

AJAX 이외에도 서버에 네트워크 요청을 보내고 정보를 받아올 수 있는 방법은 다양합니다.

그중 이번 챕터에선 모던하고 다재다능한 `fetch()` 메서드에 대해 소개해드리려 합니다. `fetch()` 는 구식 브라우저에선 지원하진 않지만(폴리필을 쓰면 사용 가능) 대부분의 모던 브라우저가 지원합니다.

`fetch()` 기본 문법은 다음과 같습니다.

```
let promise = fetch(url, [options])
```

- **`url`** – 접근하고자 하는 URL
- **`options`** – 선택 매개변수, method나 header 등을 지정할 수 있음

`options` 에 아무것도 넘기지 않으면 요청은 `GET` 메서드로 진행되어 `url` 로부터 콘텐츠가 다운로드 됩니다.

`fetch()` 를 호출하면 브라우저는 네트워크 요청을 보내고 프라미스가 반환됩니다. 반환되는 프라미스는 `fetch()` 를 호출하는 코드에서 사용됩니다.

응답은 대개 두 단계를 거쳐 진행됩니다.

먼저, 서버에서 응답 헤더를 받자마자 `fetch` 호출 시 반환받은 `promise` 가 내장 클래스 Response의 인스턴스와 함께 이행 상태가 됩니다.

이 단계는 아직 본문(body)이 도착하기 전이지만, 개발자는 응답 헤더를 보고 요청이 성공적으로 처리되었는지 아닌지를 확인할 수 있습니다.

네트워크 문제나 존재하지 않는 사이트에 접속하려는 경우같이 HTTP 요청을 보낼 수 없는 상태에선 프라미스는 거부상태가 됩니다.

HTTP 상태는 응답 프로퍼티를 사용해 확인할 수 있습니다.

- **status** – HTTP 상태 코드(예: 200)
- **ok** – 불린 값. HTTP 상태 코드가 200과 299 사이일 경우 `true`

예시:

```
let response = await fetch(url);

if (response.ok) { // HTTP 상태 코드가 200~299일 경우
  // 응답 몬문을 받습니다(관련 메서드는 아래에서 설명).
  let json = await response.json();
} else {
  alert("HTTP-Error: " + response.status);
}
```

두 번째 단계에선 추가 메서드를 호출해 응답 본문을 받습니다.

`response` 에는 프라미스를 기반으로 하는 다양한 메서드가 있습니다. 이 메서드들을 사용하면 다양한 형태의 응답 본문을 처리할 수 있습니다.

- **response.text()** – 응답을 읽고 텍스트를 반환합니다,
- **response.json()** – 응답을 JSON 형태로 파싱합니다,
- **response.formData()** – 응답을 `FormData` 객체 형태로 반환합니다. `FormData` 에 대한 자세한 내용은 다음 챕터에서 다루겠습니다.
- **response.blob()** – 응답을 Blob(타입이 있는 바이너리 데이터) 형태로 반환합니다.
- **response.arrayBuffer()** – 응답을 ArrayBuffer(바이너리 데이터를 로우 레벨 형식으로 표현한 것) 형태로 반환합니다.
- 이 외에도 `response.body` 가 있는데, ReadableStream 객체인 `response.body` 를 사용하면 응답 본문을 청크 단위로 읽을 수 있습니다. 자세한 용례는 곧 살펴보겠습니다.

지금까지 배운 내용을 토대로 GitHub에서 마지막 커밋을 JSON 객체 형태로 받아봅시다.

```
let url = 'https://api.github.com/repos/javascript-tutorial/ko.javascript.info/co
let response = await fetch(url);

let commits = await response.json(); // 응답 본문을 읽고 JSON 형태로 파싱함

alert(commits[0].author.login);
```

위 예시를 `await` 없이 프라미스만 사용하면 다음과 같이 바꿀 수 있습니다.

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commit
```

```
  .then(response => response.json())
  .then(commits => alert(commits[0].author.login));
```

응답을 텍스트 형태로 얻으려면 `.json()` 대신 `await response.text()` 를 사용하면 됩니다.

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.j

let text = await response.text(); // 응답 본문을 텍스트 형태로 읽습니다.

alert(text.slice(0, 80) + '...');
```

이번엔 `fetch` 를 사용해 fetch 명세서 우측 상단에 있는 로고(바이너리 데이터)를 가져와 보겠습니다. 참고로 `Blob` 에 대한 자세한 내용은 링크에서 살펴볼 수 있습니다.

```
let response = await fetch('/article/fetch/logo-fetch.svg');

let blob = await response.blob(); // 응답을 Blob 객체 형태로 다운로드받습니다.

// 다운로드받은 Blob을 담을 <img>를 만듭니다.
let img = document.createElement('img');
img.style = 'position:fixed;top:10px;left:10px;width:100px';
document.body.append(img);

// 이미지를 화면에 보여줍니다.
img.src = URL.createObjectURL(blob);

setTimeout(() => { // 3초 후 이미지를 숨깁니다.
  img.remove();
  URL.revokeObjectURL(img.src);
}, 3000);
```

중요:

본문을 읽을 때 사용되는 메서드는 딱 하나만 사용할 수 있습니다.

`response.text()` 를 사용해 응답을 얻었다면 본문의 콘텐츠는 모두 처리 된 상태이기 때문에 `response.json()` 은 동작하지 않습니다.

```
let text = await response.text(); // 응답 본문이 소비됩니다.
let parsed = await response.json(); // 실패
```

응답 헤더

응답 헤더는 `response.headers` 에 맵과 유사한 형태로 저장됩니다.

맵은 아닙니다. 하지만 맵과 유사한 메서드를 지원하죠. 이 메서드들을 사용하면 헤더 일부만 추출하거나 헤더 전체를 순회할 수 있습니다.

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.j

// 헤더 일부를 추출
alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

// 헤더 전체를 순회
for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

요청 헤더

`headers` 옵션을 사용하면 `fetch` 에 요청 헤더를 설정할 수 있습니다. `headers` 엔 아래와 같이 다양한 헤더 정보가 담긴 객체를 넘기게 됩니다.

```
let response = fetch(protectedUrl, {
  headers: {
    Authentication: 'secret'
  }
});
```

그런데 `headers` 를 사용해 설정할 수 없는 헤더도 있습니다. 금지된 헤더 전체 목록은 링크에서 확인할 수 있습니다.

- `Accept-Charset` , `Accept-Encoding`
- `Access-Control-Request-Headers`
- `Access-Control-Request-Method`
- `Connection`
- `Content-Length`
- `Cookie` , `Cookie2`
- `Date`
- `DNT`
- `Expect`
- `Host`
- `Keep-Alive`
- `Origin`
- `Referer`
- `TE`
- `Trailer`

- `Transfer-Encoding`
- `Upgrade`
- `Via`
- `Proxy-*`
- `Sec-*`

이런 제약은 HTTP를 목적에 맞고 안전하게 사용할 수 있도록 하려고 만들어졌습니다. 금지 목록에 있는 헤더는 브라우저만 배타적으로 설정, 관리할 수 있습니다.

POST 요청

`GET` 이외의 요청을 보내려면 추가 옵션을 사용해야 합니다.

- **`method`** – HTTP 메서드(예: `POST`)
- **`body`** – 요청 본문으로 다음 항목 중 하나이어야 합니다.
  - 문자열(예: JSON 문자열)
  - `FormData` 객체 – `form/multipart` 형태로 데이터를 전송하기 위해 쓰입니다.
  - `Blob` 나 `BufferSource` – 바이너리 데이터 전송을 위해 쓰입니다.
  - URLSearchParams – 데이터를 `x-www-form-urlencoded` 형태로 보내기 위해 쓰이는데, 요즘엔 잘 사용하지 않습니다.

대부분은 JSON을 요청 본문에 실어 보내게 됩니다.

`user` 객체를 본문에 실어 보내는 예시를 살펴봅시다.

```
let user = {
  name: 'John',
  surname: 'Smith'
};

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
```

`POST` 요청을 보낼 때 주의할 점은 요청 본문 이 문자열일 때 `Content-Type` 헤더가 `text/plain;charset=UTF-8` 로 기본 설정된다는 점입니다.

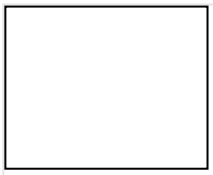
하지만 위 예시에선 JSON을 전송하고 있기 때문에 `headers` 에 제대로 된 `Content-Type` 인 `application/json` 을 설정해 주었습니다.

이미지 전송하기

`Blob` 이나 `BufferSource` 객체를 사용하면 `fetch` 로 바이너리 데이터를 전송할 수 있습니다.

예시를 살펴봅시다. `<canvas>` 에 마우스를 움직여 이미지를 만들고 '전송' 버튼을 눌러 이미지를 서버에 전송해 보겠습니다.

```
<body style="margin:0">
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>

  <input type="button" value="전송" onclick="submit()">

  <script>
    canvasElem.onmousemove = function(e) {
      let ctx = canvasElem.getContext('2d');
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    };

    async function submit() {
      let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));
      let response = await fetch('/article/fetch/post/image', {
        method: 'POST',
        body: blob
      });

      // 전송이 잘 되었다는 응답이 오고 이미지 사이즈가 얼럿창에 출력됩니다.
      let result = await response.json();
      alert(result.message);
    }

  </script>
</body>
```



이번엔 `Content-Type` 헤더를 명시적으로 설정하지 않은 점에 주목해주시기 바랍니다. `Blob` 객체는 내장 타입을 갖기 때문에 특별히 `Content-Type` 를 설정하지 않아도 됩니다. 예시는 이미지를 전송하기 때문에 `toBlob` 에 의해 `image/png` 가 자동으로 설정되었습니다. 이렇게 `Blob` 객체의 경우 해당 객체의 타입이 `Content-Type` 헤더의 값이 됩니다.

위 예시의 함수 `submit()` 을 `async/await` 없이 작성하면 아래와 같습니다.

```
function submit() {
  canvasElem.toBlob(function(blob) {
    fetch('/article/fetch/post/image', {
```

```
    method: 'POST',
    body: blob
  })
    .then(response => response.json())
    .then(result => alert(JSON.stringify(result, null, 2)))
}, 'image/png');
}
```

요약

일반적인 `fetch` 요청은 두 개의 `await` 호출로 구성됩니다.

```
let response = await fetch(url, options); // 응답 헤더와 함께 이행됨
let result = await response.json(); // json 본문을 읽음
```

물론 `await` 없이도 요청을 보낼 수 있습니다.

```
fetch(url, options)
  .then(response => response.json())
  .then(result => /* 결과 처리 */)
```

응답 객체의 프로퍼티는 다음과 같습니다.

- `response.status` – 응답의 HTTP 코드
- `response.ok` – 응답 상태가 200과 299 사이에 있는 경우 `true`
- `response.headers` – 맵과 유사한 형태의 HTTP 헤더

응답 본문을 얻으려면 다음과 같은 메서드를 사용하면 됩니다.

- **`response.text()`** – 응답을 텍스트 형태로 반환함
- **`response.json()`** – 응답을 파싱해 JSON 객체로 변경함
- **`response.formData()`** – 응답을 `FormData` 객체 형태로 반환(form/multipart 인코딩에 대한 내용은 다음 챕터에서 다룸)
- **`response.blob()`** – 응답을 Blob(타입이 있는 바이너리 데이터) 형태로 반환
- **`response.arrayBuffer()`** – 응답을 ArrayBuffer(바이너리 데이터를 로우 레벨로 표현한 것) 형태로 반환

지금까지 배운 `fetch` 옵션은 다음과 같습니다.

- `method` – HTTP 메서드
- `headers` – 요청 헤드가 담긴 객체(제약 사항이 있음)
- `body` – 보내려는 데이터(요청 본문)로 `string` 이나 `FormData`, `BufferSource`, `Blob`, `UrlSearchParams` 객체 형태

이어지는 챕터에선 이 외의 옵션과 다양한 `fetch` 유스 케이스를 살펴보겠습니다.

과제

Fetch users from GitHub

Create an async function `getUsers(names)`, that gets an array of GitHub logins, fetches the users from GitHub and returns an array of GitHub users.

The GitHub url with user information for the given `USERNAME` is: `https://api.github.com/users/USERNAME`.

There's a test example in the sandbox.

Important details:

1. There should be one `fetch` request per user.
2. Requests shouldn't wait for each other. So that the data arrives as soon as possible.
3. If any request fails, or if there's no such user, the function should return `null` in the resulting array.

테스트 코드가 담긴 샌드박스를 열어 정답을 작성해보세요.

해답

이전 주제 다음 주제

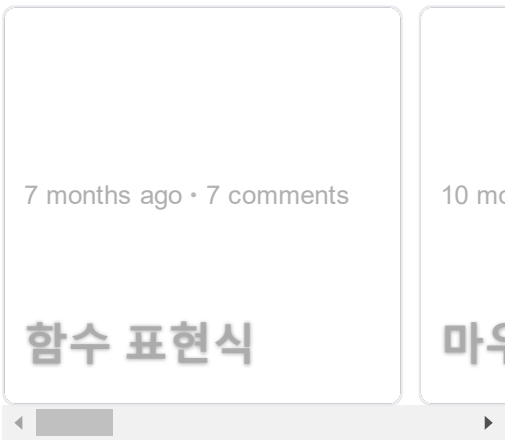
공유

튜토리얼 지도

댓글

- 추가 코멘트, 질문 및 답변을 자유롭게 남겨주세요. 개선해야 할 것이 있다면 댓글 대신 이슈를 만들어주세요.
- 잘 이해되지 않는 부분은 구체적으로 언급해주세요.
- 댓글에 한 줄짜리 코드를 삽입하고 싶다면 `<code>` 태그를, 여러 줄로 구성된 코드를 삽입하고 싶다면 `<pre>` 태그를 이용하세요. 10줄 이상의 코드는 plnkr, JSBin, codepen 등의 샌드박스를 사용하세요.

ALSO ON KO.JAVASCRIPT.INFO



Comments Community

1 Login

Recommend

Tweet Share

Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

HyoungJune Kim
• 6 months ago
와.... 잘봤습니다!
1 • Reply • Share ›

rudde • 5 months ago
await를 사용하기 위해선, async안에 있어야 한다고 배웠습니다(여기 사이트에서 async/await에서!)
근데, 해당 챕터의 예제를 보면 async없이 await가 단독으로 최상위 레벨코드에서 사용됨을 확인할 수 있는데, 이것은 어떤 경우라서 await가 단독으로 쓰인건가요?