0

비동기처리와 Promise

목차

Promise

비동기 처리

- 비동기 작업 : 특정 코드의 로직이 끝날 때까지 기다리지 않고, 나머지 코드를 먼저 실행
- 비동기 작업을 순차적으로 실행하기 위해 JavaScript에서는 Callback 함수를 사용

```
function getData() {
        var tableData;
        $.get('https://domain.com/products/1', function(response) {
                tableData = response;
        });
        return tableData;
}

console.log(getData()); // undefined
```

콜백 함수(Callback Function) : 다른 함수에 인수로 전달되는 함수
콜백 함수는 호출되기 전까지는 실행되지 않으며, 특정 이벤트(예: 데이터 처리 완료, 타이머 만료 등)가 발생했을 때 호출됩니다.

Promise

- **비동기 처리**

```
console.log('Hello’); // #1

setTimeout(function() {
    console.log('Bye');
}, 3000); // #2

console.log('Hello Again’); // #3
```

setTimeout() 역시 비동기 방식으로 실행되기 때문에 3초를 기다렸다가 다음 코드를 수행하는 것이 아니라 setTimeout()을 실행하고 나서 바로 다음 코드인 console.log('Hello Again');를 실행

Promise

비동기 처리

Callback Hell은 콜백 함수가 중첩된 형태로 여러 개의 비동기 작업을 수행할 때 발생하는 문제를 의미합니다.
콜백 함수가 중첩되면서 코드가 비정상적으로 깊어지고, 가독성이 떨어지며 유지보수가 어려워지는 현상
비동기 작업을 순차적으로 수행해야 할 때 콜백 지옥이 자주 발생합니다.

```
function firstTask(callback) {
    setTimeout(() => {
        console.log("First task done");
        callback();
    }, 1000);
}

function secondTask(callback) {
    setTimeout(() => {
        console.log("Second task done");
        callback();
    }, 1000);
}
```

```
function thirdTask(callback) {
    setTimeout(() => {
        console.log("Third task done");
        callback();
    }, 1000);
}

// 콜백 헬 형태의 코드
firstTask(() => {
    secondTask(() => {
        thirdTask(() => {
            console.log("All tasks are done");
        });
    });
});
```

Promise

- **Callback 함수로 비동기 처리 문제점 해결**
  - 콜백 함수를 사용하면 특정 로직이 끝났을 때 원하는 동작을 실행시킬 수 있습니다.

```
function getData( callbackFunc ) {
      $.get('https://domain.com/products/1',  function(response) {
            callbackFunc(response);
          // 서버에서 받은 데이터 response를 callbackFunc() 함수에 넘겨줌
        });
}

getData(function(tableData) {
      console.log(tableData); // $.get()의 response 값이 tableData에 전달됨
});
```

Promise

Callback hell

- Callback hell은 비동기 처리 로직을 위해 Callback함수를 연속해서 사용할 때 발생하는 문제입니다

```
$.get('url', function(response) {
      parseValue(response, function(id) {
            auth(id, function(result) {
                  display(result, function(text) {
                        console.log(text);
                  });
            });
      });
});
```

서버에서 데이터를 받아와 화면에 표시하기까지 인코딩, 사용자 인증 등을 처리해야 하는 경우,
모든 과정을 비동기로 처리하면 콜백 안에 콜백을 계속 포함하는 형식으로 코딩을 하게 됩니다.
코드 구조는 가독성도 떨어지고 로직을 변경하기도 어렵습니다.

Promise

- **Callback hell 해결 방법**
  - Promise나 Async를 사용
  - 코딩 패턴으로만 콜백 지옥을 해결하려면 각 콜백 함수를 분리해줍니다.

```
function parseValueDone(id) {
        auth(id, authDone);
}

function authDone(result) {
        display(result, displayDone);
}

function displayDone(text) {
        console.log(text);
}

$.get('url', function(response) {
        parseValue(response, parseValueDone);
});
```

Promise

Promise

- 비동기 작업을 처리하고 관리하기 위해 사용되는 객체
- 비동기 함수가 반환하는 객체로서 함수의 성공 또는 실패 상태를 알려줌
- 콜백 지옥(Callback Hell)을 해결하고 코드의 가독성을 높입니다
- 비동기 작업의 에러를 쉽게 처리할 수 있습니다.
- then을 통해 여러 비동기 작업을 순차적으로 처리할 수 있습니다.

```
function getData(callbackFunc) {
  $.get('url 주소/products/1', function(response) {
    callbackFunc(response);
  });
}

getData(function(tableData) {
  console.log(tableData);
});
```

Promise

Promise

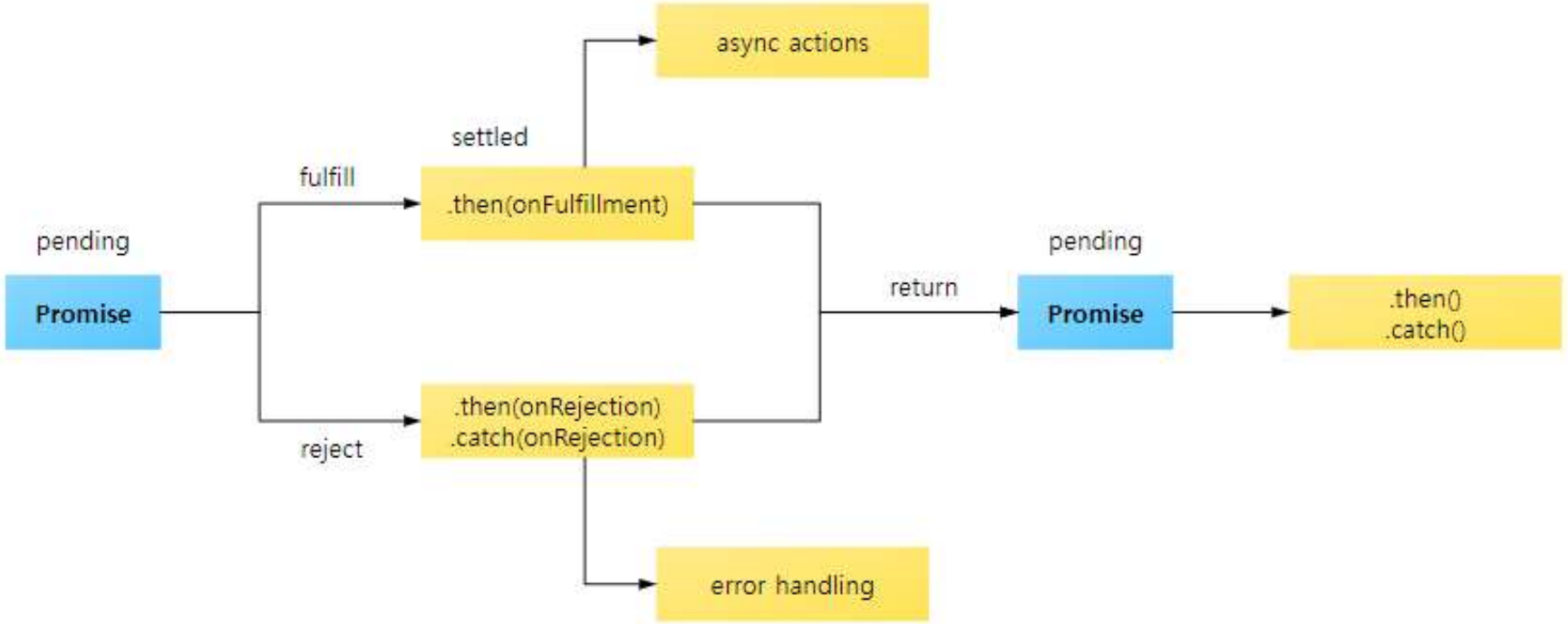
- 콜백을 직접 호출하는 방법 대신, Promise로 콜백을 부를 수 있습니다
- Promise를 사용하면 비동기 처리 시점, 비동기 함수의 결과를 쉽게 확인할 수 있고 에러 발생 위치 파악이 용이
- new Promise() 메서드를 호출할 때 콜백 함수를 선언할 수 있고, 콜백 함수의 인자는 resolve, reject입니다.

```
function getData(callback) {
  return new Promise(function(resolve, reject) {
    $.get('url 주소/products/1', function(response) {
      resolve(response);
    });
  });
}

// getData()의 실행이 끝나면 호출되는 then()
getData().then(function(tableData) {
  // resolve()의 결과 값이 여기로 전달됨
 // $.get()의 reponse 값이 tableData에 전달됨
  console.log(tableData);
});
```

Promise

- Promise states



Promise

Promise states

- Pending(대기) : Promise가 생성되고, 아직 결과가 결정되지 않은 상태입니다.
- 초기 상태로, 비동기 작업이 진행 중입니다.
- Fulfilled(이행됨) : 비동기 작업이 성공적으로 완료된 상태
- Fulfilled 상태에서는 작업의 결과 값을 반환합니다.
- Rejected (거부됨) : 비동기 작업이 실패한 상태
- Rejected 상태에서는 에러 이유를 반환합니다.

```
const promise = new Promise((resolve, reject) => {
  // 비동기 작업 수행
  const success = true;

  if (success) {
    resolve("작업 성공!"); // Fulfilled 상태로 변경
  } else {
    reject("작업 실패!"); // Rejected 상태로 변경
  }
});
//resolve(value) : 비동기 작업이 성공적으로 완료되었음을 알리고, 결과 값을 전달
//reject(reason) : 비동기 작업이 실패했음을 알리고, 실패 이유를 전달합니다.
```

Promise

Promise states

- 이행 상태가 되면 then()을 이용하여 처리 결과 값을 받을 수 있습니다.

```
function getData() {
  return new Promise(function(resolve, reject) {
    var data = 100;
    resolve(data);
  });
}
// resolve()의 결과 값 data를 resolvedData로 받음
getData().then(function(resolvedData) {
  console.log(resolvedData); // 100
});
```

- Rejected(실패) : 비동기 처리가 실패하거나 오류가 발생한 상태
- reject를 호출하면 실패(Rejected) 상태가 됩니다.

```
new Promise(function(resolve, reject) {
  resolve();
});
```

Promise

- Promise states
  - 실패 상태가 되면 실패한 이유(실패 처리의 결과 값)를 catch()로 받을 수 있습니다.

```
function getData() {
  return new Promise(function(resolve, reject) {
    reject(new Error("Request is failed"));
  });
}

// reject()의 결과 값 Error를 err에 받음
getData().then().catch(function(err) {
  console.log(err); // Error: Request is failed
});
```

Promise

- then과 catch를 사용한 처리
  - then(onFulfilled) : Promise가 Fulfilled 상태가 되었을 때 실행됩니다.
  - 작업의 결과를 인수로 받습니다.
  - catch(onRejected) : Promise가 Rejected 상태가 되었을 때 실행됩니다.
  - 실패 이유를 인수로 받습니다..

```
function getData() {
  return new Promise(function(resolve, reject) {
    reject(new Error("Request is failed"));
  });
}

// reject()의 결과 값 Error를 err에 받음
getData() .then((result) => {
    console.log(result); // 작업 성공!
  })
  .catch((error) => {
    console.error(error); // 작업 실패!
  });
```

Promise

- finally로 공통 작업 수행
  - finally(callback) : 성공 여부와 상관없이 Promise 작업이 끝난 후 항상 실행됩니다.

```
promise
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
  })
  .finally(() => {
    console.log("작업 완료!");
  });
```

Promise

- Promise Chaining
  - 프로미스는 여러 개의 프로미스를 연결하여 사용할 수 있습니다
  - then() 메서드를 호출하고 나면 새로운 프로미스 객체가 반환됩니다

```
function getData() {
  return new Promise({
    // ...
  });
}

// then() 으로 여러 개의 프로미스를 연결한 형식
getData()
  .then(function(data) {
    // ...
  })
  .then(function() {
    // ...
  })
  .then(function() {
    // ...
  });
```

Promise

- Promise Chaining

```
new Promise(function(resolve, reject){
  setTimeout(function() {
    resolve(1);
  }, 2000);
})
.then(function(result) {
  console.log(result); // 1
  return result + 10;
})
.then(function(result) {
  console.log(result); // 11
  return result + 20;
})
.then(function(result) {
  console.log(result); // 31
});
```

Promise

Promise Chaining 실무 사례

- 웹 서비스에서 사용자 로그인 인증 로직에 프로미스를 여러 개 연결

```
getData(userInfo)
  .then(parseValue)
  .then(auth)
  .then(diaplay);
```

```
var userInfo = {
  id: 'test@abc.com',
  pw: '****'
};

function parseValue() {
  return new Promise({
    // ...
  });
}
function auth() {
  return new Promise({
    // ...
  });
}
function display() {
  return new Promise({
    // ...
  });
}
```

Promise

Promise 에러 처리 방법

- then()의 두 번째 인자로 에러를 처리하는 방법

```
getData().then(
  handleSuccess,
  handleError
);
```

- catch()를 이용하는 방법

```
getData().then().catch();
```

```
var userInfo = {
function getData() {
  return new Promise(function(resolve, reject) {
    reject('failed');
  });
}

// 1. then()의 두 번째 인자로 에러를 처리하는 코드
getData().then(function() {
  // ...
}, function(err) {
  console.log(err);
});

// 2. catch()로 에러를 처리하는 코드
getData().then().catch(function(err) {
  console.log(err);
});
```

Promise

❖ Promise 에러 처리 방법

- then()의 두 번째 인자로 에러를 처리하는 방법은 then()의 첫 번째 콜백 함수 내부에서 오류가 나는 경우 오류를 제대로 잡아내지 못합니다

```
// then()의 두 번째 인자로는 감지하지 못하는 오류
function getData() {
  return new Promise(function(resolve, reject) {
    resolve('hi');
  });
}

getData().then(function(result) {
  console.log(result);
  throw new Error("Error in then()"); // Uncaught (in promise) Error: Error in
then()
}, function(err) {
  console.log('then error : ', err);
});
```

Promise

❖ Promise 에러 처리 방법

- 더 많은 예외 처리 상황을 위해 프로미스의 끝에 catch()를 정의하는 것을 권장합니다

```
// catch()로 오류를 감지하는 코드
function getData() {
  return new Promise(function(resolve, reject) {
    resolve('hi');
  });
}

getData().then(function(result) {
  console.log(result); // hi
  throw new Error("Error in then()");
}).catch(function(err) {
  console.log('then error : ', err); // then error :  Error: Error in then()
});
```

Promise

Promise.all

- 여러 개의 Promise를 병렬로 실행하고, 모두 Fulfilled되었을 때 결과를 배열로 반환합니다.
- 하나라도 Rejected되면 전체가 실패합니다.

```
const promise1 = Promise.resolve(1);
const promise2 = Promise.resolve(2);

Promise.all([promise1, promise2])
  .then((results) => {
    console.log(results); // [1, 2]
  })
  .catch((error) => {
    console.error(error);
  });
```

Promise

Promise.race

- 여러 Promise 중 가장 먼저 완료된 Promise의 결과를 반환합니다.

```
const promise1 = new Promise((resolve) => setTimeout(() => resolve("1초"), 1000));
const promise2 = new Promise((resolve) => setTimeout(() => resolve("2초"), 2000));

Promise.race([promise1, promise2]).then((result) => {
  console.log(result); // "1초"
});
```

Promise

Promise.allSetted

- 모든 Promise가 완료될 때까지 기다리며, 성공과 실패를 포함한 결과를 반환합니다.

```
const promise1 = Promise.resolve("성공!");
const promise2 = Promise.reject("실패!");

Promise.allSettled([promise1, promise2]).then((results) => {
  console.log(results);
  // [
  //   { status: "fulfilled", value: "성공!" },
  //   { status: "rejected", reason: "실패!" }
  // ]
});
```

Promise

Promise.any

- 여러 Promise 중 하나라도 Fulfilled 상태가 되면 그 결과를 반환합니다.
- 모두 Rejected되었을 경우에만 에러를 반환합니다.

```
const promise1 = Promise.reject("실패!");
const promise2 = Promise.resolve("성공!");

Promise.any([promise1, promise2]).then((result) => {
  console.log(result); // "성공!"
});
```

async & await

async

- async는 function 앞에 위치합니다
- function 앞에 async를 붙이면 해당 함수는 항상 프라미스를 반환합니다
- 프라미스가 아닌 값을 반환하더라도 이행 상태의 프라미스(resolved promise)로 값을 감싸며 이행된 프라미스가 반환되도록 합니다

```
async function f() {
  return 1;
}
f().then(alert); // 1
```

- 명시적으로 프라미스를 반환하는 것도 가능

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

async & await

await

- 프라미스가 처리될 때까지 기다립니다
- await는 async 함수 안에서만 동작합니다.

```
let value = await promise;
```

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("완료!"), 1000)
  });

  let result = await promise; // 프라미스가 이행될 때까지 기다림 (*)

  alert(result); // "완료!"
}

f();
```

함수를 호출하고, 함수 본문이 실행되는 잠시 '중단'되었다가 프라미스가 처리되면 실행이 재개됩니다.
이때 프라미스 객체의 result 값이 변수 result에 할당됩니다.
실행하면 1초 뒤에 '완료!'가 출력됩니다.

async & await

await

- await는 프라미스가 처리될 때까지 함수 실행을 기다리게 만듭니다.
- 프라미스가 처리되면 그 결과와 함께 실행이 재개됩니다
- 프라미스가 처리되길 기다리는 동안엔 엔진이 다른 일(다른 스크립트를 실행, 이벤트 처리 등)을 할 수 있기 때문에, CPU 리소스가 낭비되지 않습니다.
- async 함수가 아닌데 await을 사용하면 문법 에러가 발생합니다.
- await는 최상위 레벨 코드에서 작동하지 않습니다.
- 익명 async 함수로 코드를 감싸면 최상위 레벨 코드에도 await를 사용할 수 있습니다.
- 비동기 처리 메서드가 꼭 프로미스 객체를 반환해야 await가 의도한 대로 동작합니다.

```
async function 함수명() {
  await 비동기_처리_메서드_명();
}
```

async & await

await

- 자바스크립트의 비동기 처리 코드는 콜백을 사용해야 코드의 실행 순서를 보장받을 수 있습니다.

```
function logName() {
  var user = fetchUser('domain.com/users/1');
  if (user.id === 1) {
    console.log(user.name);
  }
}
```

```
function logName() {
  // 아래의 user 변수는 위의 코드와 비교하기 위해 일부러 남겨놓았습니다.
  var user = fetchUser('domain.com/users/1', function(user) {
    if (user.id === 1) {
      console.log(user.name);
    }
  });
}
```

async & await

async & await

- 자바스크립트의 비동기 처리 코드는 콜백을 사용해야 코드의 실행 순서를 보장받을 수 있습니다.

```
// async & await 적용 후
async function logName() {
  var user = await fetchUser('domain.com/users/1');
  if (user.id === 1) {
    console.log(user.name);
  }
}
```

async & await

async & await

- 비동기 처리 메서드가 꼭 프로미스 객체를 반환해야 await가 의도한 대로 동작합니다.

```
function fetchItems() {
  return new Promise(function(resolve, reject) {
    var items = [1,2,3];
    resolve(items)
  });
}

async function logItems() {
  var resultItems = await fetchItems();
  console.log(resultItems); // [1,2,3]
}
```

await를 사용하지 않았다면 데이터를 받아온 시점에 콘솔을 출력할 수 있게 콜백 함수나 .then()등을 사용해야 합니다
async await 문법은 비동기에 대한 사고를 하지 않아도 됩니다

async & await

async & await 예외 처리

- async & await에서 예외는 try catch로 처리합니다.

```
async function logTodoTitle() {
  try {
    var user = await fetchUser();
    if (user.id === 1) {
      var todo = await fetchTodo();
      console.log(todo.title); // delectus aut autem
    }
  } catch (error) {
    console.log(error);
  }
}
```

Fetch API

Fetch API

- fetch() 함수는 첫번째 인자로 URL, 두번째 인자로 옵션 객체를 받고, Promise 타입의 객체를 반환합니다. 반환된 객체는, API 호출이 성공했을 경우에는 응답(response) 객체를 resolve하고, 실패했을 경우에는 예외(error) 객체를 reject합니다.

```
fetch(url, options)
  .then((response) => console.log("response:", response))
  .catch((error) => console.log("error:", error));
```

- 옵션(options) 객체에는 HTTP 방식(method), HTTP 요청 헤더(headers), HTTP 요청 전문(body) 등을 설정해줄 수 있습니다. 응답(response) 객체로 부터는 HTTP 응답 상태(status), HTTP 응답 헤더(headers), HTTP 응답 전문(body) 등을 읽어올 수 있습니다.
- window.fetch()로 사용되기도 합니다.

Fetch API

❖ Fetch API GET 호출

- fetch() 함수는 디폴트로 GET 방식으로 작동하고 GET 방식은 요청 전문을 받지 않기 때문에 옵션 인자가 필요가 없습니다.

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then((response) => response.json())
  .then((data) => console.log(data));
```

- 대부분의 REST API들은 JSON 형태의 데이터를 응답하기 때문에, 응답(response) 객체는 json() 메서드를 제공합니다.

Fetch API

❖ Fetch API POST 호출

- method 옵션을 POST로 지정해주고, headers 옵션을 통해 JSON 포맷을 사용한다고 알려줘야 하며, 요청 전문을 JSON 포맷으로 직렬화화여 가장 중요한 body 옵션에 설정해줍니다.

```
fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({
    title: "Test",
    body: "I am testing!",
    userId: 1,
  }),
}).then((response) => console.log(response));
```

Fetch API

Fetch API

- HTTP 파이프라인을 구성하는 요청과 응답 등의 요소를 JavaScript에서 접근하고 조작할 수 있는 인터페이스를 제공합니다
- Fetch API가 제공하는 전역 fetch() 메서드로 네트워크의 리소스를 쉽게 비동기적으로 취득할 수도 있습니다.
- 콜백 기반 API인 XMLHttpRequest와 달리, Fetch API는 서비스 워커에서도 쉽게 사용할 수 있는 프로미스 기반으로 개선된 API입니다.

```
async function logJSONData() {
  const response = await fetch("http://example.com/movies.json");
  const jsonData = await response.json();
  console.log(jsonData);
}
```

fetch()는 가져오고자 하는 리소스의 경로를 나타내는 하나의 인수만 받습니다. 응답은 Response 객체로 표현되며, JSON 응답 본문을 바로 반환하지는 않습니다. Response는 HTTP 응답 전체를 나타내는 객체로, JSON 본문 콘텐츠를 추출하기 위해서는 json() 메서드를 호출해야 합니다. json()은 응답 본문 텍스트를 JSON으로 파싱한 결과로 이행하는, 또 다른 프로미스를 반환합니다.

Fetch API

Fetch API

- fetch() 메서드의 선택적 두 번째 매개변수는 init 객체를 사용하여 여러가지 설정을 제어할 수 있습니다.

```
// POST 메서드 구현 예제
async function postData(url = "", data = {}) {    // 옵션 기본 값은 *로 강조
  const response = await fetch(url, {
    method: "POST",    // *GET, POST, PUT, DELETE 등
    mode: "cors",    // no-cors, *cors, same-origin
    cache: "no-cache", // *default, no-cache, reload, force-cache, only-if-cached
    credentials: "same-origin", // include, *same-origin, omit
    headers: {
      "Content-Type": "application/json",
      // 'Content-Type': 'application/x-www-form-urlencoded',
    },
    redirect: "follow", // manual, *follow, error
    referrerPolicy: "no-referrer", // no-referrer, *no-referrer-when-downgrade, origin, origin-
when-cross-origin, same-origin, strict-origin, strict-origin-when-cross-origin, unsafe-url
    body: JSON.stringify(data), // body의 데이터 유형은 반드시 "Content-Type" 헤더와 일치
해야 함
  });
  return response.json(); // JSON 응답을 네이티브 JavaScript 객체로 파싱
}
```

Fetch API

Fetch API

- fetch() 작업을 취소하려면 AbortController와 AbortSignal 인터페이스를 사용합니다

```
const controller = new AbortController();
const signal = controller.signal;
const url = "video.mp4";

const downloadBtn = document.querySelector("#download");
const abortBtn = document.querySelector("#abort");

downloadBtn.addEventListener("click", async () => {
  try {
    const response = await fetch(url, { signal });
    console.log("다운로드 완료", response);
  } catch (error) {
    console.error(`다운로드 오류: ${error.message}`);
  }
});

abortBtn.addEventListener("click", () => {
  controller.abort();
  console.log("다운로드 중단됨");
});
```