Javascript 심화

비동기 프로그래밍

김태민

저번에 우리는

함수형 코드

렉시컬 스코프

변수의 접근 가능 범위를 정의하는 개념

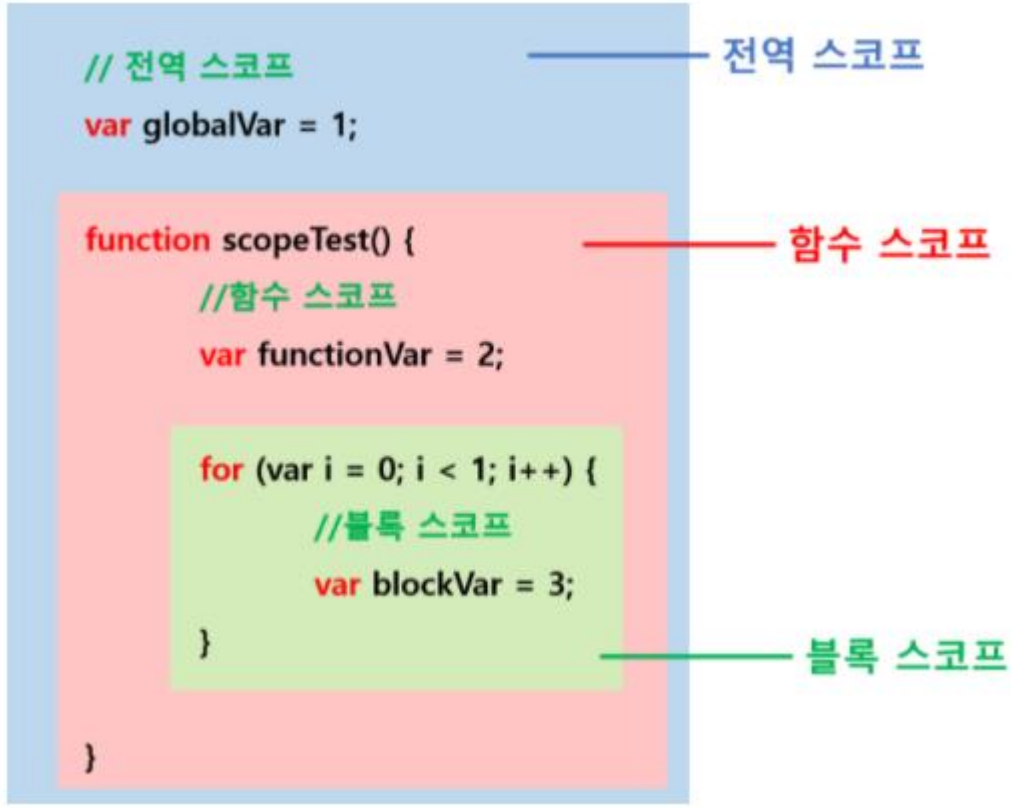
자바스크립트는 **렉시컬 스코프(정적 스코프)**를 사용

변수가 어디서 선언되었는지에 따라 접근 가능 여부가 결정

함수를 어디서 호출했는지가 아니라

함수를 어디에 정의했는지에 따라

상위 스코프를 결정



고차함수 조합

```
const numbers = [1, 2, 3, 4, 5];
const result = numbers
    .filter(num => num % 2 === 0)
    .map(num => num * 2)
    .reduce((acc, curr) => acc + curr, 0);
console.log(result); // 12 (2*2 + 4*2)
```

실습 환경

실습환경

OS : Window / Mac

브라우저 : Chrome

에디터 : VS Code https://code.visualstudio.com/

VS Code 익스텐션 : ESLint, Prettier, HTML CSS Support, HTML to CSS autocompletion, Auto Rename Tag, Auto Close Tag, htmltagwrap

Git : git bash, github 가입

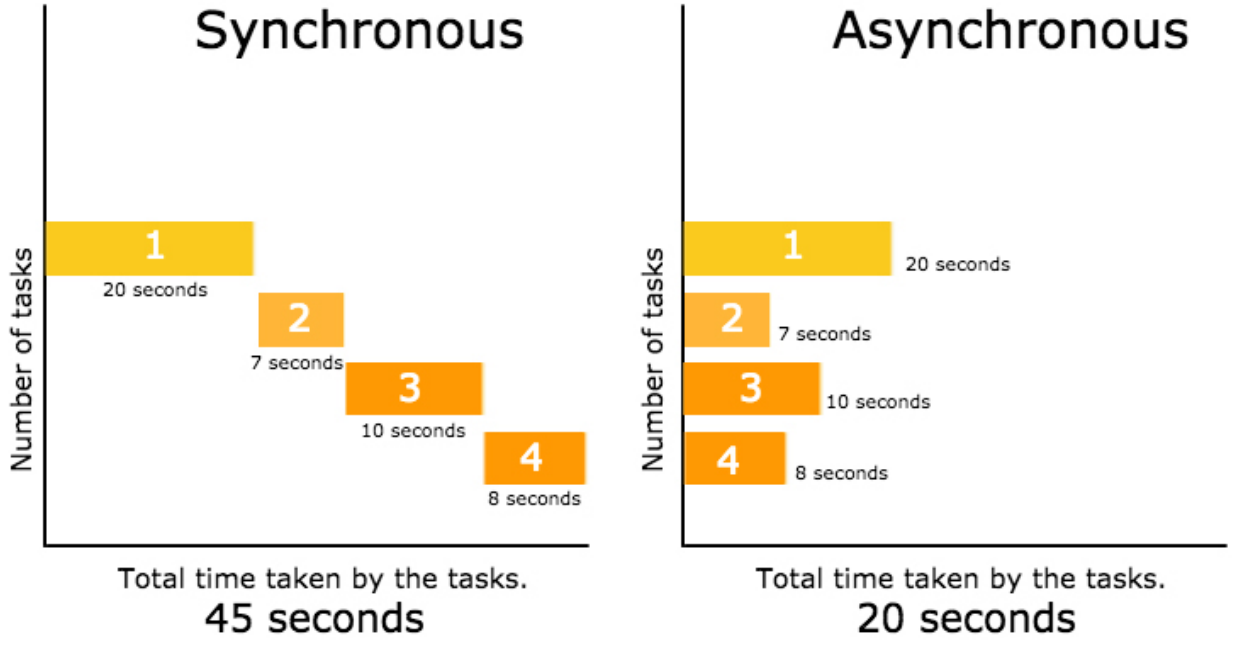
Etc : postman

비동기 프로그래밍

동기와 비동기 개념

동기 : 동기는 순차적으로 일이 진행 되는 것

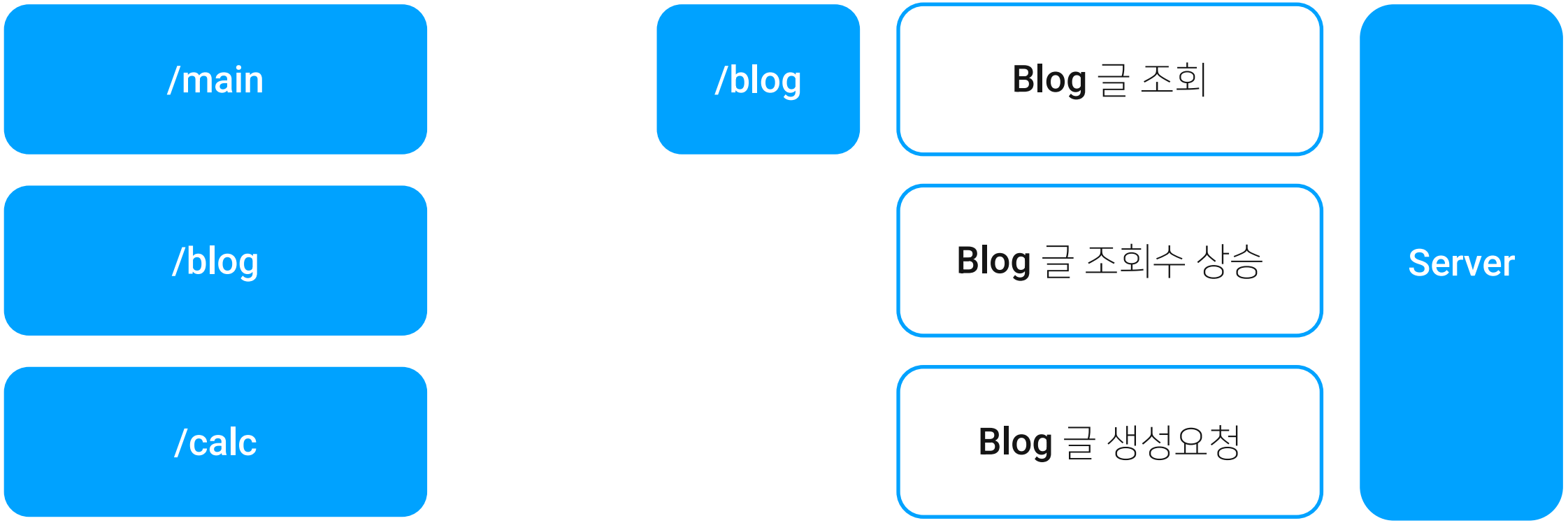
비동기 : 작업이 완료되기를 기다리지 않고 다음 작업을 실행



웹 개발에서 동기와 비동기

동기(Sync)로 페이지 이동

비동기(Async)로 데이터 조회, 가공, 표현



콜백함수

다른 함수의 인자로 전달되어 특정 작업이 완료된 후 호출되는 함수.

비동기 작업의 결과를 처리하거나, 특정 이벤트 발생 시 실행.

자바스크립트에서 비동기 처리의 기본 도구.

예: 버튼 클릭 시 실행되는 이벤트 핸들러, 타이머 완료 후 호출되는 함수.

```
function sayHello(name, callback) {
    console.log(`안녕, ${name}!`);
    callback();
}
sayHello("학생", () => console.log("콜백 실행!"));
```

콜백지옥

비동기 작업을 처리하기 위해 콜백 함수를 깊게 중첩하면 코드가 복잡해지는 현상.

- 들여쓰기 증가로 가독성 저하.
- 에러 처리 복잡, 디버깅 어려움.
- 복잡한 로직: 순차적 작업이 많아질수록 관리 어려움.

```
getUser(user => {
  getPosts(user, posts => {
    getComments(posts, comments => {
      console.log(comments);
    });
  });
});
```

Promise chaining

.then()으로 성공 결과 처리, .catch()로 에러 처리.

체이닝으로 순차적 비동기 작업 관리.

```
new Promise((resolve) => {
  setTimeout(() => resolve(1), 1000);
})
  .then(result => {
    console.log(result);
    return result + 1;
  })
  .then(result => console.log(result))
  .catch(error => console.log("에러:", error));
```

Promise.all로 병렬 처리

Promise.all: 여러 Promise를 병렬로 실행, 모두 완료될 때까지 대기.

모든 Promise가 성공하면 결과 배열 반환, 하나라도 실패하면 즉시 에러

```
const p1 = new Promise(resolve => setTimeout(() => resolve("작업 1"), 1000));
const p2 = new Promise(resolve => setTimeout(() => resolve("작업 2"), 2000));
Promise.all([p1, p2]).then(results => console.log(results));
```

Async / Await

async: 함수가 Promise를 반환하도록 설정.

await: Promise가 완료될 때까지 대기, 동기처럼 코드 작성 가능.

Promise 대비 가독성과 유지보수성 향상.

```
async function fetchData() {
    let promise = new Promise(resolve => setTimeout(() => resolve("데이터 가져옴!"), 2000));
    let result = await promise;
    console.log(result);
}
fetchData();
```

Async / Await 으로 Fetch Api 간소화

https://www.notion.so/fetch-api-229caf5650aa80bc80bdf20b790f5b04

```
let response = await fetch("https://jsonplaceholder.typicode.com/posts/2");
let data = await response.json();
console.log(data.title);
```

Fetch API

Fetch API

API (Application Programming Interface):

서로 다른 소프트웨어 시스템이 데이터를 주고받거나 기능을 호출할 수 있도록 정의된 인터페이스

UI (User Interface):

사용자가 소프트웨어와 상호작용하는 접점(예: 버튼, 입력 폼)

GUI (Graphical User Interface):

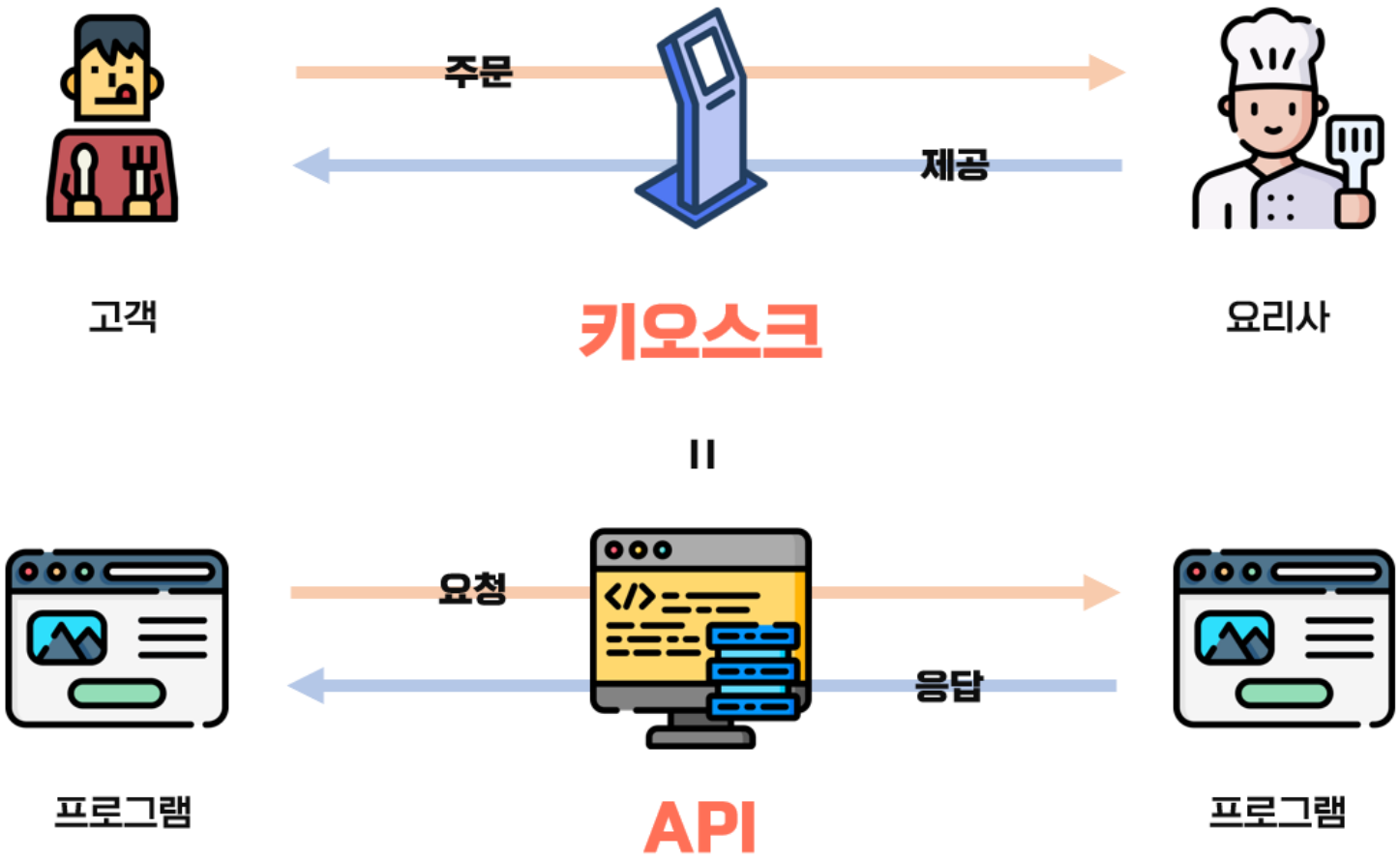
그래픽 요소(아이콘, 창)를 사용한 UI(예: 웹사이트, 모바일 앱)

CLI (Command Line Interface):

텍스트 기반 명령어로 소프트웨어와 상호작용(예: 터미널에서 git 명령)

Interface

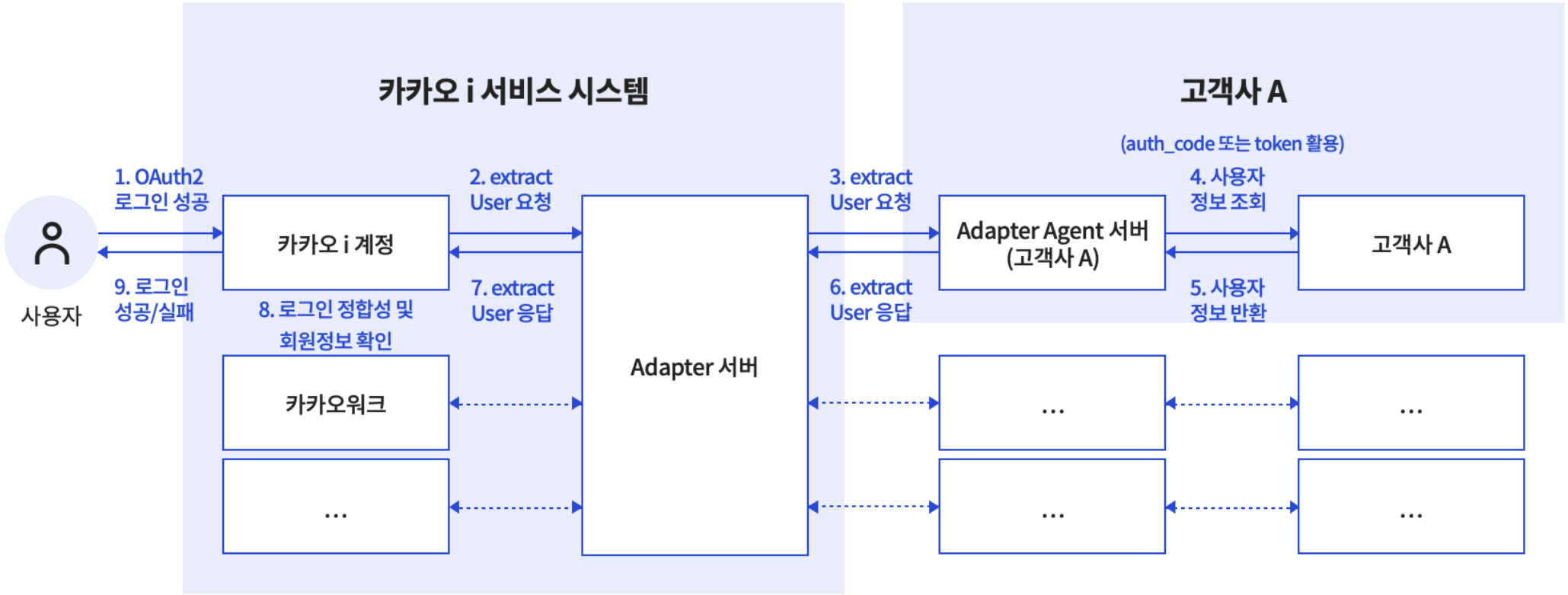
Interface



Fetch API

[사용자]가 **[카카오 I 서비스 시스템]의 API** 규칙에 맞게 "로그인"을 요청

[카카오 I 서비스 시스템]이 **[고객사A]의 API** 규칙에 에 맞게 사용자 정보를 요청



Fetch API

웹 브라우저에서 HTTP 요청(GET, POST 등)을 보내고 서버 API로부터 응답을 받는 현대적인 방법

Promise 기반: .then()과 .catch()로 비동기 처리

기존 XMLHttpRequest 대비 간결하고 직관적

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

Fetch API 데이터 요청

GET 요청: 서버 API에서 데이터 가져오기

URL 구성: 엔드포인트와 쿼리 파라미터(예: ?userId=1)

응답 객체: response.ok, response.status

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(response => {
    if (!response.ok) throw new Error("요청 실패");
    return response.json();
  })
  .then(data => console.log(data.title))
  .catch(error => console.error(error));
```

Fetch API로 POST 요청

POST 요청: 서버 API에 데이터 전송

옵션 객체: method, headers, body

```
fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ title: "새 포스트", body: "내용", userId: 1 })
})
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Fetch 응답 처리

응답 데이터 처리: JSON, 텍스트, Blob 등

상태 코드 확인: 200(성공), 404(찾을 수 없음), 500(서버 에러)

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(response => response.json())
  .then(data => {
    document.getElementById("output").innerText = data.title;
  });
```

Fetch+Async/Await

Fetch + Async/Await

Fetch 응답 처리

Fetch를 Async/Await으로 표현

Async 함수: Promise를 반환하는 함수

Await: Promise가 해결될 때까지 기다림

```
async function fetchPost() {
  const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
  const data = await response.json();
  return data.title;
}
fetchPost().then(title => console.log(title));
```

Fetch + Async/Await

다중 비동기 요청 처리

여러 await 호출로 순차적 처리

병렬 처리: Promise.all()

```
async function fetchMultiple() {
  const [post, user] = await Promise.all([
    fetch("https://jsonplaceholder.typicode.com/posts/1").then(res => res.json()),
    fetch("https://jsonplaceholder.typicode.com/users/1").then(res => res.json())
  ]);
  console.log(post.title, user.name);
}
fetchMultiple();
```

Fetch + Async/Await

에러 핸들링

Async/await에서의 에러 처리 필요성

Try...catch로 에러 핸들링

```
async function fetchWithError() {
    try {
        const response = await fetch("https://invalid-url");
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error("에러 발생:", error.message);
    }
}
fetchWithError();
```

Fetch + Async/Await

복잡한 데이터 fetch

특정 사용자(userId: 2)의

포스트와 사용자 정보를 가져와,

HTML에 이름과 포스트 제목을 표시

```
async function displayUserPosts() {
  try {
    const [user, posts] = await Promise.all([
      fetch("https://jsonplaceholder.typicode.com/users/2").then(res => res.json()),
      fetch("https://jsonplaceholder.typicode.com/posts?userId=2").then(res => res.json())
    ]);
    document.getElementById("output").innerHTML = `<h2>${user.name}</h2>`;
    const ul = document.createElement("ul");
    posts.forEach(post => {
      const li = document.createElement("li");
      li.textContent = post.title;
      ul.appendChild(li);
    });
    document.getElementById("output").appendChild(ul);
  } catch (error) {
    console.error("에러:", error);
  }
}
displayUserPosts();
```

Web Storage

Web Storage

WebStorage

Web Storage: 브라우저에서 클라이언트 측 데이터를 저장하는 API(localStorage, sessionStorage)

Cookie: 서버와 클라이언트 간 데이터 교환(예: 인증 토큰), HTTP 요청에 자동 포함

localStorage: 영구 저장, 동일 출처 내 모든 탭/윈도우 공유

sessionStorage: 탭 세션 동안만 유지, 탭 간 공유 불가

IndexedDB: 대용량 데이터, 비동기, 구조화된 데이터 저장

Cookie

키-값 쌍, HTTP 헤더에 포함되어 서버와 데이터 교환

특징: 용량 제한(~4KB), 만료일 설정 가능, 문자열만 저장, 도메인/경로 제한

사용 사례: 사용자 인증(세션 ID), 추적(광고), 사용자 선호도 저장

제한: 보안 문제(CSRF, XSS), HTTP 요청 오버헤드

LocalStorage

동일 출처 내 영구 저장, 모든 탭/윈도우 공유

주요 메서드: setItem(), getItem(), removeItem(), clear()

사용 사례:

- 사용자 설정 저장(예: 테마, 언어)
- 폼 데이터 저장(새로고침 후 복원)
- 캐싱(예: API 응답 저장)

제한: 동기 API, 보안 민감 데이터 부적합

SessionStorage

동일 탭 내 세션 동안만 유지, 탭 간 공유 불가

주요 메서드: setItem(), getItem(), removeItem(), clear()

사용 사례:폼 데이터 임시 저장(새로고침 시 복원)

페이지 네비게이션 상태(예: 필터 설정)

일회성 데이터(예: CSRF 토큰)

제한: 탭 종료 시 데이터 삭제, 동기 API

JSON과 WebStorage

Web Storage는 문자열 저장

객체는 JSON.stringify()로 직렬화하여 저장 가능

데이터 복원: JSON.parse()로 객체 변환

```
const user = { name: "Jane", preferences: { theme: "dark" } };
localStorage.setItem("user", JSON.stringify(user));
const storedUser = JSON.parse(localStorage.getItem("user"));
console.log(storedUser.name, storedUser.preferences.theme);
```

오늘 우리는

Fetch API

API (Application Programming Interface):

서로 다른 소프트웨어 시스템이 데이터를 주고받거나 기능을 호출할 수 있도록 정의된 인터페이스

UI (User Interface):

사용자가 소프트웨어와 상호작용하는 접점(예: 버튼, 입력 폼)

GUI (Graphical User Interface):

그래픽 요소(아이콘, 창)를 사용한 UI(예: 웹사이트, 모바일 앱)

CLI (Command Line Interface):

텍스트 기반 명령어로 소프트웨어와 상호작용(예: 터미널에서 git 명령)

Fetch + Async/Await

복잡한 데이터 fetch

특정 사용자(userId: 2)의

포스트와 사용자 정보를 가져와,

HTML에 이름과 포스트 제목을 표시

```
async function displayUserPosts() {
  try {
    const [user, posts] = await Promise.all([
      fetch("https://jsonplaceholder.typicode.com/users/2").then(res => res.json()),
      fetch("https://jsonplaceholder.typicode.com/posts?userId=2").then(res => res.json())
    ]);
    document.getElementById("output").innerHTML = `<h2>${user.name}</h2>`;
    const ul = document.createElement("ul");
    posts.forEach(post => {
      const li = document.createElement("li");
      li.textContent = post.title;
      ul.appendChild(li);
    });
    document.getElementById("output").appendChild(ul);
  } catch (error) {
    console.error("에러:", error);
  }
}
displayUserPosts();
```

Web Storage

WebStorage

Web Storage: 브라우저에서 클라이언트 측 데이터를 저장하는 API(localStorage, sessionStorage)

Cookie: 서버와 클라이언트 간 데이터 교환(예: 인증 토큰), HTTP 요청에 자동 포함

localStorage: 영구 저장, 동일 출처 내 모든 탭/윈도우 공유

sessionStorage: 탭 세션 동안만 유지, 탭 간 공유 불가

IndexedDB: 대용량 데이터, 비동기, 구조화된 데이터 저장

감사합니다