브라우저 저장소 + 쿠키

브라우저 저장소는 웹 브라우저에서 데이터를 저장하는 데 사용되는 여러 기술을 말합니다.

이 저장소들은 사용자가 웹 사이트를 이용할 때 필요한 정보를 로컬에서 보관하여 사용자 경험을 개선하고, 오프라인에서도 데이터 접근을 가능하게 합니다. 대표적으로 localStorage, sessionStorage, 그리고 IndexedDB가 있습니다.

▼ localStorage

장점

- 웹 브라우저를 닫아도 데이터가 사라지지 않으며 영구적으로 저장됩니다.
- 쉬운 API로 데이터 저장 및 접근이 가능합니다.

단점

- 브라우저 저장소의 데이터가 웹 서버로 자동 전송되지 않는다.

  ▼ 보충 설명

  HTTP 쿠키와 대조적으로, localStorage와 sessionStorage에 저장된 데이터는 웹 페이지에서 명시적으로 API를 호출하여 사용할 때만 접근할 수 있습니다.

  웹 스토리지 기술(localStorage와 sessionStorage)과 HTTP 쿠키는 서버와 클라이언트 간의 데이터 교환에서 다르게 작동합니다.

  - **HTTP 쿠키:**

    HTTP 쿠키는 클라이언트와 서버 사이의 작은 데이터 조각을 저장하며, 이 데이터는 HTTP 요청을 통해 자동으로 서버에 전송됩니다.

    즉, 사용자가 웹사이트에 접속할 때마다 쿠키 정보가 자동적으로 요청 헤더에 포함되어 서버로 전송되어, 서버가 사용자를 인식하거나 사용자의 세션을 관리할 수 있게 합니다.

    ▼ 사용자의 세션 관리

    "사용자 세션을 관리할 수 있다"는 표현은 웹 애플리케이션에서 사용자의 상태나 활동을 일정 기간 동안 추적하고 유지하는 것을 의미합니다. 이는 사용자가 웹 사이트에 로그인했을 때 그 상태를 유지하고, 사용자의 활동을 연속적으로 연결해주는 방식으로 작동합니다. 세션 관리는

웹 애플리케이션에서 중요한 역할을 하며, 사용자 경험과 보안을 모두 증진시키는 데 기여합니다.

사용자 세션 관리의 주요 기능:

1. **사용자 인증 유지**

사용자가 로그인하면, 웹 서버는 그 사용자의 인증 상태를 세션에 저장합니다. 이후 사용자가 다른 페이지로 이동하거나 요청을 할 때, 서버는 세션 데이터를 참조하여 사용자가 이미 인증되었는지 확인합니다. 이 방법으로 사용자가 사이트 내에서 로그인 상태를 유지할 수 있습니다.

2. **일시적인 데이터 저장**

세션은 사용자가 웹 사이트를 이용하는 동안 필요한 일시적인 정보를 저장하는 데 사용됩니다. 예를 들어, 온라인 쇼핑을 할 때 장바구니에 담은 상품 목록이 세션에 저장되어 다른 페이지로 이동해도 그 내용이 유지됩니다.

3. **보안성 향상**

세션 관리는 사용자의 민감한 정보를 안전하게 보호하는 데 중요한 역할을 합니다. 예를 들어, 세션 ID는 고유하며 예측이 불가능해야 하고, HTTP 쿠키에 저장될 때는 안전하게 전송되어야 합니다. 이런 방식으로 세션 하이재킹과 같은 보안 위협을 방지할 수 있습니다.

4. **사용자 활동 추적**

웹 사이트는 사용자가 어떤 페이지를 방문하고, 어떤 상품을 봤는지 등의 활동을 세션을 통해 추적할 수 있습니다. 이 정보는 사용자 경험을 개선하거나 마케팅 전략을 수립하는 데 사용될 수 있습니다.

세션 관리 기술:

- **HTTP 쿠키(서버&클라이언트 사이드)**: 사용자의 브라우저에 세션 ID를 저장하여 서버가 사용자를 인식할 수 있도록 합니다.
- **세션 저장소(서버사이드)**: 여기서 말하는 세션저장소는 클라이언트 기반 임시 저장소인 세션스토리지가 아닌 전통적인 세션저장소, 즉 서버 측에서 사용자의 세션 데이터를 관리하는 저장소입니다. 데이터베이스, 메모리, 파일 시스템 등 여러 형태로 구현될 수 있습니다.

사용자 세션을 관리하는 것은 웹 개발에서 매우 중요한 부분으로, 사용자에게 안전하고 일관된 서비스를 제공하는 데 필수적입니다.

▼ 세션저장소

전통적으로 말하는 '세션 저장소'와 클라이언트 사이드의 sessionStorage는 다른 개념입니다. 이 두 가지를 구분하는 것이 중요합니다.

전통적인 세션 저장소 (Server-Side Session Storage)

- **서버 기반**: 이는 서버 측에서 세션 데이터를 저장하는 저장소입니다. 이 저장소는 메모리, 데이터베이스, 또는 파일 시스템 등 다양한 방식으로 구현될 수 있습니다.
- **데이터 보안**: 사용자의 세션 데이터는 서버에 저장되므로, 데이터 보안과 무결성 관리가 더욱 중요하고 철저히 이루어집니다.
- **세션 ID 관리**: 사용자가 웹 사이트에 접속할 때 생성되는 세션 ID는 클라이언트에 보내지고, 이후 모든 HTTP 요청에 포함되어 서버에 다시 전송됩니다. 서버는 이 세션 ID를 사용하여 사용자의 세션 데이터에 접근합니다.

클라이언트 사이드 sessionStorage

- **클라이언트 기반**: sessionStorage는 웹 브라우저 내에서 작동하며, 사용자의 장치에 데이터를 저장합니다. 각 브라우저 탭에 독립적으로 데이터가 저장되며, 탭을 닫으면 데이터가 사라집니다.
- **임시 데이터 저장**: sessionStorage는 사용자가 브라우저 탭을 닫기 전까지만 데이터를 유지합니다. 이는 주로 탭이나 창 기반의 짧은 세션에서 사용자의 상태를 유지하는 데 사용됩니다.
- **직접적인 데이터 접근**: 서버와의 통신 없이 웹 페이지의 JavaScript를 통해 데이터에 접근하고 조작할 수 있습니다.

이렇게 서버 사이드 세션 저장소는 세션 데이터의 중앙집중적 관리를 통해 사용자 인증과 상태 관리의 일관성을 유지하는 데 중점을

둡니다. 반면, sessionStorage는 사용자의 브라우저 내에서만 데이터를 관리하고, 주로 임시 데이터 저장용으로 사용되어, 각 사용자의 탭 또는 세션 독립성을 보장합니다. 두 기술은 사용 목적과 데이터 관리 방식에서 크게 다르므로, 적절한 상황에 맞게 선택하여 사용하는 것이 중요합니다.

- **localStorage와 sessionStorage:**

  반면, localStorage와 sessionStorage는 데이터를 웹 브라우저의 저장 공간에 보다 명시적으로 저장합니다. 이 데이터는 서버로 자동 전송되지 않습니다. 웹 페이지의 JavaScript 코드를 통해서만 접근할 수 있으며, 데이터를 서버로 전송하고자 할 때는 개발자가 직접적으로 API를 호출하여 이를 구현해야 합니다.

  예를 들어, AJAX 요청을 만들거나 API를 통해 서버에 데이터를 전송하는 과정에서 localStorage나 sessionStorage에서 데이터를 가져와 명시적으로 포함시켜야 합니다.

```
// 예를 들어, 사용자 정보를 localStorage에 저장
localStorage.setItem('userInfo', JSON.stringify
({ name: 'John', age: 30 }));

// 서버에 사용자 정보를 전송할 필요가 있을 때
fetch('https://example.com/api/user', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: localStorage.getItem('userInfo')  //
명시적으로 localStorage에서 정보를 가져와서 전송
});
```

  이 예제에서 볼 수 있듯, localStorage에서 데이터를 가져와 서버에 전송하는 것은 완전히 개발자의 의도에 따라 수행됩니다.

이러한 차이는 데이터를 처리하는 방법에서 중요한 보안 및 설계 고려사항을 야기합니다.

쿠키는 CSRF(Cross-Site Request Forgery) 공격에 취약할 수 있으나, localStorage나 sessionStorage는 이러한 종류의 공격으로부터 상대적으로 더 안전합니다. 데이터가 자동으로 서버로 전송되지 않기 때문에, 애플리케이션 개발자는 어떤 데이터가 서버로 전송되는지 더 세밀하게 통제할 수 있습니다.

▼ 단점으로 간주될 수 있는 이유

1. 서버와의 자동 동기화 부재

- **상태 동기화의 복잡성**: localStorage나 sessionStorage는 클라이언트 측에서만 데이터를 관리합니다. 웹 애플리케이션에서 사용자의 상태나 데이터를 서버와 일관되게 유지해야 하는 경우, 개발자는 모든 변경 사항을 수동으로 서버에 전송하도록 코드를 작성해야 합니다. 이는 추가적인 개발 노력과 복잡성을 요구하며, 실수로 데이터 동기화를 놓치거나 오류를 만들 수 있는 여지를 제공합니다.

2. 실시간 업데이트의 부족

- **다중 세션/장치 간 동기화 문제**: 사용자가 여러 기기나 브라우저 탭에서 동일한 서비스를 사용할 경우, localStorage나 sessionStorage에 저장된 데이터는 자동으로 다른 세션 또는 장치와 동기화되지 않습니다. 예를 들어, 한 탭에서 로그인 상태를 변경했을 때 다른 탭이나 기기에서 이 변경 사항을 인식하지 못할 수 있습니다. 이는 사용자 경험을 저하시킬 수 있습니다.

3. 서버 기반 기능의 제한

- **스케일링과 데이터 관리의 어려움**: 서버에서 사용자의 데이터를 관리하고 분석하는 것이 더 용이한 경우가 많습니다. 예를 들어, 데이터 분석, 백업, 또는 다른 시스템과의 통합을 통해 보다 풍부한 인사이트와 기능을 제공할 수 있습니다. localStorage나 sessionStorage를 사용하면 이러한 서버 기반 기능을 활용하기 위해 추가적인 데이터 동기화 작업이 필요합니다.

4. 보안과 데이터 무결성

- **보안 및 데이터 무결성 문제**: 클라이언트 측에서만 데이터를 처리하게 되면, 보안과 데이터 무결성을 유지하는 데 제한이 따릅니다. 서버에서 데이터를

검증하고 처리하는 것이 보안을 강화하고 데이터 오류를 줄이는 데 도움이 될 수 있습니다.

- 5MB 정도의 저장 공간 제한이 있습니다.
- 동기적 API만 제공하여 큰 데이터를 처리할 때 페이지 렌더링이 느려질 수 있습니다.

예시

localStorage는 주로 사용자의 선호도나 로그인 상태 같은 정보를 장기간 보관할 때 사용됩니다. 다음은 localStorage에 데이터를 저장하고, 불러오며, 삭제하는 기본적인 사용 방법입니다.

```
// 데이터 저장
localStorage.setItem('username', 'JohnDoe');

// 데이터 불러오기
var username = localStorage.getItem('username');
console.log(username);  // "JohnDoe"

// 데이터 삭제
localStorage.removeItem('username');

// 모든 데이터 삭제
localStorage.clear();
```

▼ sessionStorage

장점

- 브라우저 탭이나 창이 열려 있는 동안 데이터가 유지되고, 탭/창을 닫으면 데이터가 사라집니다.
- 쉬운 API로 데이터 저장 및 접근이 가능합니다.

단점

- 데이터가 세션 동안에만 유지되며 영구적인 저장이 불가능합니다.
- localStorage와 같이 5MB의 저장 공간 제한이 있습니다.

- 동기적 API 사용으로 대량의 데이터 처리 시 성능 문제가 발생할 수 있습니다.

예시

sessionStorage는 주로사용자 세션 동안 일시적으로 필요한 데이터를 저장할 때 사용됩니다. 예를 들어, 페이지간에 사용자 입력을 임시 저장할 때 유용합니다.

```
// 데이터 저장
sessionStorage.setItem('sessionName', 'ExampleSession');

// 데이터 불러오기
var sessionData = sessionStorage.getItem('sessionName');
console.log(sessionData);  // "ExampleSession"

// 데이터 삭제
sessionStorage.removeItem('sessionName');

// 모든 데이터 삭제
sessionStorage.clear();
```

▼ IndexedDB

IndexedDB는 클라이언트 측에서 작동하는 웹 브라우저 내장 데이터베이스입니다. 이는 사용자의 브라우저 내에 로컬로 데이터를 저장하며, 웹 애플리케이션에 복잡한 데이터 구조를 저장하고 비동기적으로 처리할 수 있는 기능을 제공합니다.

장점

- 복잡한 데이터를 저장하기에 적합하며, 큰 용량의 데이터 저장이 가능합니다.
    - 키-값 저장 방식을 넘어서, 객체와 배열 같은 복잡한 데이터 구조를 저장할 수 있으며, 인덱스를 통한 효율적인 데이터 검색과 조작이 가능합니다.
- 비동기 API를 제공하여 대량 데이터 처리 시 브라우저의 응답성을 유지할 수 있습니다.
    - 데이터베이스 작업은 비동기적으로 수행되어, UI(사용자 인터페이스)의 응답성을 저하시키지 않습니다.
- 트랜잭션 지원으로 데이터 무결성을 보장합니다.

    ▼ 트랜잭션

트랜잭션은 데이터베이스 관리 시스템에서 중요한 개념으로, 하나 이상의 연산을 그룹화하여 단일 작업 단위로 처리하는 방법을 말합니다. 트랜잭션은 데이터베이스의 무결성을 보장하는 데 필수적인 역할을 합니다.

모든 연산이 성공적으로 완료되면 트랜잭션은 "커밋"되어 변경사항이 데이터베이스에 영구적으로 반영됩니다. 만약 트랜잭션 중 하나라도 실패하면, "롤백"이 발생하여 트랜잭션의 모든 변경사항이 취소되고 데이터베이스는 트랜잭션 실행 이전 상태로 복원됩니다.

트랜잭션의 주요 특징은 다음과 같은 **네 가지 원칙(ACID)**에 따라 설계됩니다:

1. **원자성(Atomicity)**:
    - 트랜잭션 내의 모든 연산은 하나의 단위로 처리되어야 합니다. 모든 연산이 성공적으로 완료되거나, 하나라도 실패할 경우 전체 트랜잭션이 취소됩니다.
2. **일관성(Consistency)**:
    - 트랜잭션이 실행되기 전과 후의 데이터베이스 상태는 모두 일관된 상태를 유지해야 합니다. 즉, 트랜잭션이 데이터베이스의 모든 제약조건을 준수해야 하며, 어떤 시점에서든 데이터베이스는 유효한 상태여야 합니다.
3. **독립성(Isolation)**:
    - 여러 트랜잭션이 동시에 실행될 때, 각 트랜잭션은 서로 독립적으로 실행되어야 합니다. 다른 트랜잭션의 중간 결과에 영향을 받지 않아야 합니다.
4. **지속성(Durability)**:
    - 트랜잭션이 일단 커밋되면, 그 결과는 시스템의 장애가 발생하더라도 영구적으로 반영되어야 합니다. 즉, 데이터가 안전하게 저장되어야 합니다.

트랜잭션의 예시:

예를 들어 은행 계좌 이체는 트랜잭션으로 처리됩니다. 사용자 A가 사용자 B에게 돈을 이체할 때, A의 계좌에서 돈이 빠져나가고 B의 계좌에 돈이 들어가는 두 가지 작업이 성공적으로 완료되어야 합니다. 만약 이 중 한 작업이라도 실패한다면, 두 작업 모두 취소되어야 하며 계좌의 원래 상태를 유지해야 합니다.

트랜잭션은 데이터베이스 시스템뿐만 아니라, 다양한 시스템에서 데이터의 일관성과 무결성을 유지하는 데 중요한 역할을 합니다.

▼ 데이터 무결성(data integrity)

데이터베이스의 무결성(data integrity)이란 데이터가 정확하고, 일관되게 관리되고, 신뢰할 수 있는 상태를 유지하는 것을 말합니다. 무결성이 유지되는 데이터베이스는 오류가 없다기보다는, 데이터가 일정한 규칙과 표준을 준수하고, 예상치 못한 방법으로 데이터가 손상되거나 잘못 변경되지 않고 시스템이 의도한 대로 올바르게 작동한다는 의미를 포함합니다.

단점

- API 사용이 복잡하여 배우기가 다소 어려울 수 있습니다.
- 브라우저 지원이 일관성이 없을 수 있습니다.

예시

IndexedDB는 복잡한 구조의 데이터를 저장하고 검색할 필요가 있을 때 사용됩니다. 예를 들어, 사용자의 복잡한 설정 정보나 오프라인에서 사용할 데이터베이스를 관리할 때 적합합니다.

```
// IndexedDB 데이터베이스 생성 및 열기
var request = window.indexedDB.open("MyTestDatabase",
1);

request.onerror = function(event) {
  console.error("Database error: " + event.target.errorC
ode);
};

request.onupgradeneeded = function(event) {
  var db = event.target.result;

  // 객체 저장소 생성
  var objectStore = db.createObjectStore("name", { keyPa
```

```
th: "id" });

  // 인덱스 생성
  objectStore.createIndex("name", "name", { unique: fals
e });
};

request.onsuccess = function(event) {
  var db = event.target.result;

  // 데이터 추가
  var customerData = [
    { id: "1", name: "John Doe", age: 30 },
    { id: "2", name: "Jane Doe", age: 25 }
  ];

  var transaction = db.transaction(["name"], "readwrit
e");

  transaction.oncomplete = function(event) {
    console.log("All done!");
  };

  transaction.onerror = function(event) {
    console.error("Transaction error!");
  };

  var objectStore = transaction.objectStore("name");
  customerData.forEach(function(customer) {
    var request = objectStore.add(customer);
    request.onsuccess = function(event) {
      console.log("Customer added: ", customer.name);
    };
  });
};
```

이 코드는 IndexedDB에서 데이터베이스를 열고, 객체 저장소를 생성하며, 데이터를 추가하는 기본적인 절차를 보여줍니다.

▼ HTTP 쿠키

HTTP 쿠키, 종종 웹 쿠키라고도 불리는 것은 웹 서버가 사용자의 웹 브라우저에 전송하는 작은 데이터 조각입니다. 이 데이터는 사용자의 컴퓨터에 저장되며, 사용자가 동일한 웹사이트를 재방문할 때 브라우저에 의해 서버로 다시 전송됩니다.

따라서 쿠키는 서버와 클라이언트 사이드 모두에서 사용됩니다.

쿠키는 일반적으로 **작은 크기의 데이터를 클라이언트-서버 간에 지속적으로 유지해야 할 때 사용**되며, 주로 사용자 인증 정보(세션 ID 등)를 저장하는 데 적합합니다.

쿠키의 주된 목적은 웹사이트가 사용자를 "기억"할 수 있도록 하여, 더 나은 사용자 경험을 제공하는 것입니다.

쿠키의 종류

1. **세션 쿠키**
    - 웹 브라우저가 열려 있는 동안에만 유효하고, 브라우저를 닫으면 자동으로 삭제됩니다. 일시적인 데이터 저장에 사용됩니다.
2. **영구 쿠키**
    - 설정된 만료 기간이 지나거나 사용자가 수동으로 삭제할 때까지 유지됩니다. 사용자의 장기적인 선호 설정을 저장하는 데 사용됩니다.
3. **써드파티 쿠키**
    - 방문한 웹사이트가 아닌 다른 도메인에서 생성하고 관리하는 쿠키입니다. 주로 광고 네트워크에 의해 설정되며, 사용자 행동을 추적하는 데 사용됩니다.

서버 사이드에서의 역할

1. **세션 관리**: 로그인, 사용자 인증, 사용자 세션 상태 등을 유지합니다. 서버는 사용자가 로그인할 때 쿠키에 세션 ID를 저장하고 클라이언트에 전송합니다. 이후 클라이언트는 모든 HTTP 요청에 이 쿠키를 포함시켜 서버가 각 요청이 같은 사용자로부터 온 것임을 인식하게 합니다.
2. **개인 설정**: 사용자의 선호도, 테마 설정 등을 저장하여 사용자 맞춤형 경험을 제공합니다. 다음 방문 때 같은 환경을 제공합니다.

3. **트래킹**: 사용자의 웹사이트 방문 패턴과 행동을 추적합니다. 이 정보는 마케팅 전략을 개선하고 사용자 경험을 개인화하는데 사용될 수 있습니다.

클라이언트 사이드에서의 역할

1. **데이터 저장**: 쿠키는 클라이언트의 브라우저에 저장되어, 서버로부터 받은 정보를 유지합니다. 사용자가 웹사이트에 다시 접속할 때, 이 쿠키 정보가 서버로 전송되어 사용자의 설정이나 로그인 상태를 바로 복원할 수 있게 합니다.
2. **접근성**: 쿠키는 웹 브라우저를 통해 관리될 수 있으며, 사용자 또는 클라이언트 측 스크립트에서도 접근할 수 있습니다.

쿠키는 이렇게 서버와 클라이언트 양쪽에서 상호 작용하는 메커니즘이며, **서버에서 설정하고 클라이언트에서 저장하고 다시 서버로 보내지는 방식**으로 운용됩니다.

그러나 보안 측면에서 쿠키의 사용에는 주의가 필요하며, 서버와 클라이언트 사이에서 쿠키 데이터를 안전하게 전송하고 관리하는 것이 중요합니다.

쿠키의 보안과 개인정보 보호

쿠키는 편리하고 유용한 도구이지만, 사용자의 개인 정보와 관련하여 몇 가지 보안 우려가 있습니다. 예를 들어, 쿠키는 사용자의 브라우징 활동을 추적할 수 있으며, 쿠키 데이터가 제3자에 의해 접근되거나 수정될 위험이 있습니다. 또한 쿠키는 CSRF(Cross-Site Request Forgery) 공격에 취약할 수 있습니다.

이를 해결하기 위해, 웹사이트는 HTTPS를 통한 안전한 데이터 전송, HttpOnly 및 Secure 플래그 설정을 통해 쿠키의 안전을 강화할 수 있습니다. 또한, 많은 국가에서는 사용자의 동의 없이 쿠키를 사용하는 것을 제한하는 법적 규정을 두고 있습니다.

▼ CSRF(Cross-Site Request Forgery, 사이트 간 요청 위조)

CSRF는 사용자가 인지하지 못한 상태에서, 사용자의 인증 정보를 이용해 악의적인 요청을 보내는 웹 보안 공격입니다. CSRF 공격은 주로 사용자가 웹 애플리케이션에 로그인한 상태에서 이루어지며, 공격자가 사용자의 권한을 이용해 원하지 않는 행동을 수행하게 만듭니다.

CSRF 공격의 원리

1. **사용자가 로그인**:
    - 사용자가 A 사이트에 로그인하여 세션 쿠키를 받습니다. 이 쿠키는 사용자가 로그인한 상태를 유지하는 데 사용됩니다.

2. **사용자가 악의적인 사이트 방문**:
    - 사용자가 A 사이트에 로그인한 상태에서, 악의적인 사이트 B를 방문합니다. 이 사이트는 사용자가 모르게 CSRF 공격을 준비합니다.
3. **악의적인 요청 전송**:
    - 사이트 B는 사용자가 A 사이트에 로그인한 세션을 이용하여, A 사이트에 요청을 보냅니다. 이 요청은 사용자의 브라우저를 통해 전송되므로, A 사이트는 이를 합법적인 요청으로 인식합니다.

CSRF 공격의 예시

예를 들어, 사용자가 은행 사이트에 로그인한 상태에서 악성 사이트를 방문하면, 악성 사이트는 사용자의 은행 계좌에서 돈을 이체하는 요청을 보낼 수 있습니다. 사용자는 이를 알지 못한 채 공격에 당하게 됩니다.

CSRF 방지 방법

CSRF 공격을 방지하기 위해 다양한 방법이 사용됩니다:

1. **CSRF 토큰 사용**:
    - 각 사용자 세션마다 고유한 CSRF 토큰을 생성하고, 이를 폼 제출 시에 포함시킵니다. 서버는 요청 시 CSRF 토큰을 검증하여 유효성을 확인합니다. 토큰이 일치하지 않으면 요청을 거부합니다.

```
<!-- 예시 HTML 폼 -->
<form action="/transfer" method="POST">
    <input type="hidden" name="csrf_token" value
="서버에서 생성한 CSRF 토큰">
    <!-- 다른 입력 필드 -->
    <button type="submit">이체</button>
</form
```

2. **SameSite 쿠키 속성**:
    - SameSite 쿠키 속성을 설정하여, 동일 사이트 내에서만 쿠키가 전송되도록 제한합니다. 이는 CSRF 공격을 방지하는데 효과적입니다.

```
Set-Cookie: sessionid=abcd1234; SameSite=Strict
```

3. **CORS(Cross-Origin Resource Sharing) 설정**:
    - 서버는 CORS 설정을 통해 어떤 도메인에서 서버 자원에 접근할 수 있는지를 제어할 수 있습니다. 이를 통해 신뢰할 수 없는 도메인의 접근을 제한합니다.
4. **Referrer 검사**:
    - 서버는 요청의 Referrer 헤더를 검사하여, 요청이 적절한 출처에서 왔는지를 확인할 수 있습니다.

▼ 저장 방식 선택

저장 방식을 선택할 때는 데이터의 유형, 보안 요구 사항, 데이터의 생명주기, 그리고 브라우저 간의 호환성 등을 고려해야 합니다.

쿠키 사용 시기

- **사용자 인증 및 세션 관리**:
    - 쿠키는 서버에서 생성된 세션 ID를 저장하는 데 주로 사용됩니다. 이 세션 ID를 통해 사용자의 로그인 상태를 유지하고, 서버에 요청을 보낼 때마다 사용자를 인증할 수 있습니다.
- **서버와 클라이언트 간의 작은 데이터 전송**:
    - 쿠키는 매 HTTP 요청과 함께 서버로 자동으로 전송되기 때문에, 서버가 필요로 하는 작은 정보(예: 사용자 선호 언어, 사용자 설정 등)를 유지하는 데 유용합니다.
- **사용자 추적 및 광고 목적**:
    - 쿠키는 사용자의 웹 사용 행태를 추적하는 데 사용되며, 이 정보는 맞춤 광고와 웹사이트의 사용자 경험 개선에 활용될 수 있습니다.

로컬스토리지 사용 시기

- **클라이언트 측에서의 데이터 저장 및 접근**:
  - 애플리케이션의 성능을 향상시키기 위해 자주 사용되는 데이터(예: 애플리케이션의 테마, 복잡한 사용자 설정)를 로컬스토리지에 저장하여 서버 요청 없이 빠르게 접근할 수 있습니다.
- **오프라인 데이터 사용**:
  - 웹 애플리케이션이 오프라인에서도 동작해야 할 경우, 로컬스토리지를 사용하여 필요한 데이터를 저장하고 접근할 수 있습니다.

세션스토리지 사용 시기

- **탭 또는 세션별 데이터 저장**:
  - 세션스토리지는 브라우저 탭이 열려 있는 동안만 데이터를 유지합니다. 사용자가 탭을 닫으면 데이터도 소멸되므로, 탭이나 세션 동안 필요한 임시 데이터(예: 양식 입력 데이터, 페이지 내의 임시 설정)를 저장하는 데 적합합니다.

보안을 염두에 두고, 민감한 데이터는 서버 측에서 관리하는 것이 안전합니다.

▼ 로그인 상태 관리

로그인 상태 관리를 위해 일반적으로 JWT와 함께 로컬스토리지 또는 쿠키를 사용합니다.

토큰 기반 인증:

- JWT(Json Web Tokens)와 같은 토큰 기반 인증 시스템은 로그인 상태를 유지하는 현대적인 방법 중 하나입니다. 사용자가 로그인하면 서버는 토큰을 생성하고 이를 사용자에게 전달합니다. 사용자는 이 토큰을 로컬스토리지, 세션스토리지 또는 쿠키에 저장할 수 있으며, 이후의 요청에 토큰을 포함시켜 서버가 사용자를 인증할 수 있게 합니다.

  ▼ JWT(JSON Web Tokens)

  JWT는 사용자 인증을 관리하는 데 널리 사용되는 방법입니다. JWT는 사용자의 인증 정보를 서버와 클라이언트 사이에서 교환할 수 있는 JSON 객체로 인코딩된 토큰 형태로 제공합니다. 이 토큰은 서명되어 있어 변조를 방지하고, 필요한 경우 토큰 자체를 암호화하여 추가적인 보안 레벨을 제공할 수 있습니다.

JWT 사용 시 고려사항:

1. **저장 위치**: JWT를 로컬스토리지나 세션스토리지에 저장할 수 있지만, XSS 공격에 취약할 수 있습니다. 이를 방지하기 위해, 가능한 쿠키에 저장하고 HttpOnly, Secure 플래그를 활성화하여 클라이언트 사이드 스크립트의 접근을 제한하는 것이 좋습니다.
2. **만료 시간**: JWT는 만료 시간을 포함할 수 있으므로, 토큰이 만료되면 사용자는 다시 인증해야 합니다. 이는 보안을 강화하는 한 방법입니다.
3. **보안 설정**: 가능한 한 HTTPS를 사용하여 통신하고, 적절한 CORS 설정을 구현하여 불필요한 사이트 간 요청을 방지해야 합니다.

▼ **로컬스토리지 이용**

사실, 로그인 상태를 로컬스토리지에 저장하는 방법은 흔하지만, 이 방법은 몇 가지 중요한 보안 문제를 야기할 수 있습니다. 로그인 상태를 안전하게 관리하는 방법은 애플리케이션의 보안 요구사항을 깊이 고려해야 합니다.

로컬스토리지의 보안 취약성:

- 로컬스토리지에 저장된 데이터는 HTTP(S)를 통해 암호화되지 않고, 사이트가 크로스 사이트 스크립팅(XSS) 공격을 받을 경우, 악의적인 스크립트에 의해 쉽게 액세스될 수 있습니다. 따라서 로그인 토큰과 같은 민감한 정보를 로컬스토리지에 저장하는 것은 위험할 수 있습니다.

보안 강화 방안

로컬스토리지를 사용하여 JWT나 다른 인증 토큰을 저장하는 경우, 몇 가지 보안 조치를 통해 위험을 줄일 수 있습니다:

1. **XSS 방지**:
   - **철저한 입력 검증 및 필터링**: XSS 공격을 방지하기 위해, 웹 애플리케이션 내의 모든 사용자 입력을 철저히 검증하고 필터링해야 합니다. 잠재적인 스크립트 삽입을 방지합니다.
   - **Content Security Policy(CSP):** CSP를 설정하여 스크립트 소스를 제한하고, XSS 공격을 예방합니다.
2. **HTTPS 사용**:
   - **전송 중인 데이터 보호**: HTTPS를 통해 통신을 암호화하여(서버와 클라이언트 간의 데이터 전송을 암호화) 네트워크 상에서 데이터를 보호합니다. 이

는 중간자 공격을 방지하는 데 도움이 됩니다.

+이는 로컬스토리지에 저장된 데이터가 아닌, 네트워크 상에서의 데이터 전송을 보호합니다. 로컬스토리지에 저장된 데이터 자체는 HTTPS에 의해 암호화되지 않습니다. HTTPS는 전송 중인 데이터를 암호화하여 중간자 공격을 방지하는 역할을 합니다. 때문에 로컬스토리지에 저장된 데이터는 여전히 클라이언트 측에서 취약할 수 있습니다.

3. **토큰 만료 및 갱신**:
   - 짧은 수명의 토큰을 사용하고, 주기적으로 토큰을 갱신하여 사용자가 오랜 시간 동안 같은 토큰을 사용하지 않도록 합니다. 이는 토큰이 탈취되더라도 그 유효 기간을 최소화하는 데 도움이 됩니다.
4. **로컬스토리지 대신 쿠키 사용**:
   - 가능한 경우, 민감한 정보는 `HttpOnly`와 `Secure` 플래그가 설정된 쿠키에 저장하는 것이 좋습니다. 이는 클라이언트 측 스크립트가 쿠키에 접근하는 것을 방지하고, HTTPS를 통해 안전하게 전송됩니다.

     예시: HttpOnly와 Secure 쿠키 설정

     ```
     Set-Cookie: token=your_jwt_token; HttpOnly; Secure
     ```

     - **HttpOnly**: 클라이언트 측 JavaScript에서 쿠키에 접근할 수 없도록 설정합니다.
     - **Secure**: 쿠키가 HTTPS를 통해서만 전송되도록 설정합니다.
     - **SameSite**: CSRF 공격을 방지하기 위해 쿠키가 동일 사이트 요청에서만 전송되도록 설정합니다.

토큰 기반 인증에서 로컬스토리지를 사용하는 것은 간편하고, 많은 개발자들이 채택하고 있지만, 보안 위험을 인지하고 적절한 조치를 취하는 것이 중요합니다. 가능하면 민감한 인증 정보를 HttpOnly와 Secure 플래그가 설정된 쿠키에 저장하고, 철저한 XSS 방지 조치를 통해 보안을 강화하는 것이 좋습니다.

▼ **쿠키 이용**

쿠키를 이용한 보안한 방법:

- 사용자 인증 정보(예: 세션 ID)를 쿠키에 저장하는 것이 일반적입니다. 쿠키는 HttpOnly와 Secure 플래그를 설정할 수 있어, 클라이언트 측 스크립트가 쿠키에 접근하는 것을 방지하고, 정보를 안전하게 전송할 수 있도록 합니다. 이는 XSS 공격과 데이터 도난의 위험을 줄여줍니다.

로그인 상태 관리의 최선의 방법 선택

- 보안이 중요한 경우, 가능한 쿠키의 HttpOnly와 Secure 옵션을 사용해야 합니다. 이렇게 설정된 쿠키는 JavaScript를 통한 접근이 제한되며 HTTPS를 통해서만 전송됩니다.
- 클라이언트에서 민감한 데이터를 처리할 때는 항상 최신 보안 관행을 따라야 하며, 데이터 보호를 위해 적절한 조치를 취해야 합니다.

결론적으로, 로그인 상태를 로컬스토리지에 저장하는 것은 간편하고 유연할 수 있지만, 보안을 위해 쿠키와 같은 더 안전한 방법을 고려하는 것이 좋습니다. 특히, 웹 애플리케이션에서 보안이 중요한 요소라면, 보다 안전한 인증 관리 방법을 선택해야 합니다.