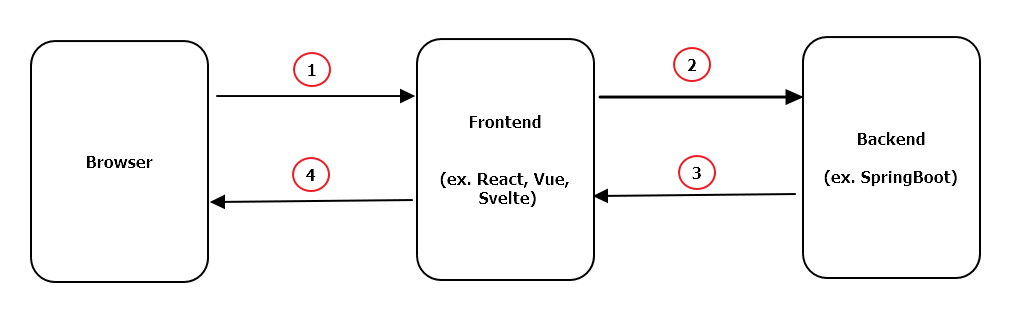
**브라우저 – 프론트엔드 – 백엔드 로그인 세션**

1. **사용한 기술**

**Frontend: Sveltekit**

**Backend: SpringBoot(Spring-Security 사용)**

1. **상황**

**내가 원하는 방향**

1. (브라우저) 로그인 화면에서 id와 password를 입력한다.
2. (프론트) 로그인 <form>태그를 통해 action=(백엔드 로그인 url), (method=post)로 쏜다.
3. (백엔드) Spring-Security에서는 백엔드 로그인 url을 가로채 인증/인가를 생성하고 새로운 세션ID를 발급하여 쿠키에 담아 프론트로 전송한다.
4. (프론트) 백엔드로부터 받은 세션ID를 브라우저에게 전송한다.
5. (브라우저) 프론트로부터 세션ID를 받고 공지사항을 들어가 나의 권한에 맞는 페이지를 띄우기 위해 Frontend에게 세션ID를 보낸다.
6. (프론트) 브라우저로부터 공지사항 url과 헤더 값에 세션ID를 받고 DB로부터 게시글을 받기 위해 Backend로 세션ID를 보낸다.
7. (백엔드) 세션ID를 확인하여 ROLE(권한)을 확인하여 로그인한 사용에게 허락된 게시글 데이터를 담아 프론트로 보낸다.
8. (프론트) 데이터를 받아 UI 처리와 사용자의 권한에 따라 글쓰기 삭제 등에 대한 UI처리를 진행하고 브라우저에게 화면을 출력할 준비를 마친다. (SSR로 진행)  
   => 해당 부분은 CSR로 처리하는 경우에만 프론트에서 백엔드에게 로그인한 사용자 세션을 유지하면서 보낼 수 있게 된다…. 이유가 뭔지 살펴보았다. 맨 아래 참고
9. (브라우저) 프론트로부터 완성된 공지사항 페이지를 보게된다. (이때부터 공지사항 내부에서 발생하는 이벤트는 CSR로 처리)

**실제 벌어진 일**

위의 상황에서 나는 6번서부터 더 이상 브라우저에 입력된 세션ID값이 사용되지 않고 프론트엔드에서 빈 값으로 데이터를 보내면 백엔드에서 새로운 세션을 만들어 버리는 것을 확인하였다.

그 결과 브라우저에서 입력된 세션 ID값이 아닌 새로운 세션 ID값이 백엔드에서 사용하게 되고 결국 사용자에 대한 정보가 **Anonymous**로 발생하게 되었다.

**해결 방법**

1. **세션(쿠키)**

현재 문제는 프론트단에서 백엔드로 세션ID를 보낼 때 빈 값으로 보내진다는 것이다.

(정확히 표현하자면 프론트단의 서버엔진이 새로운 세션 ID를 만들어서 백엔드로 전송하는 형태이다.)

이를 해결하기 위해 브라우저에 담아 보낸 세션ID(로그인 인증이 된 상태)를 유지시키면서 프론트엔드에서 백엔드까지 보내기 위해 쿠키에 세션ID를 계속 담고 있도록 공유시키는 것이다.

그러기 위해 사용되는 방법은 credentials옵션이다.

보내는 전송 메서드 마다 사용옵션은 다양하므로 각자 자기가 사용하는 메서드에 맞게 옵션을 켜주도록 하면 된다. (참고로 나는 Sveltekit에서 fetch를 사용하기에 credentials : “input”옵션으로 켜주었다.)

위 옵션을 사용하게 되면 프론트엔드에서 백엔드로 데이터를 전송할 때 Request헤더 값에 Set-Cookie 속성 값이 들어가게 되고 쿠키 안에는 JSESSIONID값이 들어가면서 브라우저에서 받은 세션ID를 정상적으로 백엔드까지 전송이 가능해진다.

따라서 나의 경우에는 6번 상황에 fetch메서드를 호출하는 구간에 credentials옵션을 통해 해결할 수 있었다.

그리고 중요한 점은 백엔드에서 세션ID를 쿠키에 담아 프론트로 전송하기 위해서는 당연히 백엔드도 credentials옵션을 켜야 한다.

(credentials 옵션을 키는 방법들은 구글링에 정보가 많고 매우 간단하여 생략한다. 다만 주의할 점으로는 백엔드에서 credentials옵션을 사용했을 때는 백엔드 설정구간 allowedOrigin, allowedHeader와 같은 속성 값에 ‘\*’옵션은 사용하지 않고 정확한 url값이나 속성에 맞는 값을 적어줘야 한다.)

참고로 나는 Sveltekit에서 개발을 했는데 form태그로 사용자 id, password로 보냈을 때는 자동으로 credentials옵션이 달려서 백엔드로 전송되는 것을 확인했다.

하지만 로그인 과정에서는 백엔드에서 어차피 새로운 세션 ID를 발급해서 주기 때문에 Spring security에서 response헤더에 쿠키 값이 담길 수 있도록 security 설정에 credentials옵션을 넣어주는 것이 포인트다.

또한 “**내가 원하는 방향**” 목차에서 3번 과정이 정상 동작했던 이유로는 credentials옵션을 게시글 참고하여 개발하다가 security 응답해주는 설정 과정에 넣어 놨었기 때문…ㅇㅅㅇ

위의 세션 저장소는 default로 톰캣 서버 내부적으로 저장된다. 따라서 내가 서버를 재부팅하게 될 시 전부 날라간다.

따라서 메모리에 저장하는 용도와 세션 공유와 같은 용도로 사용을 하고 싶다면 레디스에 세션들을 저장하는 방법도 존재한다.

(https://doing7.tistory.com/18)

해당 방법은 말 그대로 세션에 저장하는 것이기 때문에 서버측면에서는 사용자가 많아지면 서버 용량이 늘어날 수 밖에 없다.

그리고 서버가 여러 대 존재할 경우 로드밸런싱에 의해 사용자가 다른 서버로 접속하게 되면 해당 서버에 내 세션ID를 재발급 받아야되는데 이를 세션공유라는 기술을 통해 커버가 가능해진다. 하지만 세션 공유에 대한 기술력이 세션이 많아질수록 유지비용이 늘어난다고 한다….

그래서 다음으로 JWT(토큰)방법을 생각할 수 있다.

1. **JWT 토큰**

JWT 토큰은 세션을 서버에 저장하지 않고 통신 헤더에 담아 사용자 정보를 인증/인가하기 때문에 서버의 부담이 적어진다.

또한 만료 기간을 정해주게 되면 계속 사용할 수 있어 좋지만…. 아무래도 사용자의 정보를 담은상태로 네트워크가 Xss와 Csrf공격에 취약하다는 것.

세션의 경우 서버에서 사용자정보를 관리하며 쿠키에 담긴 값을 httpOnly를 통해 클라이언트의 접근을 차단시켜버릴 수 있어 보안은 좀 더 좋은 것 같다.

(XSS공격을 하더라도 cookie값이 노출이 되지 않고 나의 경우 프론트엔드 script에서 백엔드로부터 받은 쿠키 값을 출력해보려 별짓을 다해보았으나 httpOnly옵션 때문에 출력이 되지 않았다.)

가장 기본적으로 JWT 인증 구성 방식은 아래와 같다.

차트이(가) 표시된 사진

자동 생성된 설명

1. AccessToken과 Refresh토큰 저장 위치  
   (<https://it-eldorado.tistory.com/165>)   
   (<https://velog.io/@jisu2281l/TIL-Refresh-Token-%EC%A0%80%EC%9E%A5-%EC%9C%84%EC%B9%98%EB%8A%94-%EC%96%B4%EB%94%94>)

* Cookie  
  Xss 공격에 HttpOnly옵션 설정과 CSRF 공격을 막기 위해 SameSite옵션 및 Lax, Strict등의 설정도 필요하다. (추가적으로 Secure 옵션 설정을 통해 Https 프로토콜 사용할 때만 쿠키가 전송되도록 할 수 있다.)
* Local Storage  
  로컬 스토리지의 경우 매 요청마다 서버에게 자동으로 전송되는 것이 아니기 때문에 CSRF의 공격으로 안전할 수 있다. 다만 HttpOnly와 같은 보안옵션들을 설정할 수 없어 Excape처리 등으로 Xss공격에 대한 대응이 필요
* Session Storage

세션에 저장하는 경우에는 JWT 토큰의 목적과 상이한 점이 많아 대체적으로 Session 저장소를 사용하지 않는 편이다.

해당 블로그를 통해(<https://gksdudrb922.tistory.com/217#JwtTokenProvider>)

(<https://velog.io/@jkijki12/Jwt-Refresh-Token-%EC%A0%81%EC%9A%A9%EA%B8%B0#jwtproviderjava-%EC%88%98%EC%A0%95>)

JWT토큰을 기본 베이스로 개발하였고 여기서 추가적으로 알아보자.

1. JwtExceptionFilter가 필요하다. 해당 이유로는 필터링 과정에서 기존의 JwtAuthenticationFilter만 거치게 될 경우 AccessToken 만기 후에 어떠한 설정 값을 담아 클라이언트에게 응답으로 던질 수가 없게 된다. 따라서 에러응답만 응답하게 되어 403이 발생하지만 JwtExceptionFilter를 통해 Client에게 에러 처리에 대한 응답을 던짐으로써 클라이언트는 응답코드의 조건분기를 통해 처리가 가능해진다. (<https://devjem.tistory.com/72>)
2. 그리고 위의 인증 과정 그림과 같이 구현하기위해 RefreshToken은 사용자 DB에 넣어두고 AcccessToken과 Refresh Token을 프론트로 넘긴 뒤 프론트는 localStorage에 저장한다. 프론트에서는 서버 호출 시에는 AccessToken만 담아서 호출하도록 한다.
3. 만일 AccessToken이 만료되었을 시에는 1)의 과정을 토대로 만료 응답을 클라이언트에게 보낸다. 그리고 클라이언트는 AccessToken과 RefreshToken을 담아 서버로 요청을 보내면 서버에서는 RefreshToken 만료기한이 다 되었는 지 아니면 유효한 토큰인지 확인 후 DB에 저장된 토큰과 비교하여 일치했을 때 AccessToken을 재발급해준다.
4. 추가적으로 AccessToken 재발급시에 Refresh Token도 재발급해줌으로써 보안을 좀 강화했고 재발급 동시에 사용자 DB의 refreshToken값도 UPDATE해주었다.
5. 해당 과정을 개발하기위해 방향성에 대해 참고했던 블로그들  
   (<https://tansfil.tistory.com/59>)  
   (<https://llshl.tistory.com/32>)  
   (<https://nowgnas.github.io/posts/refreshtoken/>)  
   (<https://han-um.tistory.com/17>)  
   (<https://velog.io/@solchan/%EB%B0%B1%EC%97%85-Refresh-Token-%EB%B0%9C%EA%B8%89%EA%B3%BC-Access-Token-%EC%9E%AC%EB%B0%9C%EA%B8%89>)
6. 최종적으로 사용자 로그아웃의 기능에서 블랙리스트에 accessToken을 넣어야하는 부분 때문에 Redis를 사용하게 되었다. (<https://velog.io/@jbb9229/redis-introduce01>)

* Redis를 사용해야되는 이유로는 첫 번째로 inmemory 형태라 데이터의 접근이 잦을수록 유리한 장점이 존재한다.
* 따라서 accessToken을 블랙리스트에 올려두면 빠르게 찾아 확인이 가능해지나 일반 DB의 경우 칼럼을 만들고 token을 넣고 jpa를 거쳐 확인을 해야 된다.
* refreshToken의 만료시간을 정해두고 redis에 등록할 수 있지만 DB에 저장해둘 경우 주기적으로 refreshToken을 꺼내어 만료가 되었는지 확인해야된다.

(<https://sol-devlog.tistory.com/22>)

(<https://velog.io/@joonghyun/SpringBoot-Jwt%EB%A5%BC-%EC%9D%B4%EC%9A%A9%ED%95%9C-%EB%A1%9C%EA%B7%B8%EC%95%84%EC%9B%83>)

(<https://bestinu.tistory.com/54>)

(<https://wildeveloperetrain.tistory.com/61>)

**결론적으로 해당 DB외에 추가적으로 Redis가 존재하며 위 그림에 더해 아래처럼 JWT구성도와 동일하게 개발 완료**

도표이(가) 표시된 사진

자동 생성된 설명

* **참고 사항**

1) front에서는 localStorage에 저장하는 과정에서 값을 못불러오는 경우가 있는데 SSR의 형태로 메서드를 돌릴 때이다. 따라서 useEffect 메서드를 사용하거나 svelteKit의 경우 랜더링 방식을 선택할 수 있기 때문에 csr값을 true로 설정하여 돌리도록 하면 된다.

(<https://velog.io/@sji7532/Next-Next%EC%97%90%EC%84%9C%EC%9D%98-alert-window-document-localStorage-%EC%82%AC%EC%9A%A9-%EA%B0%80%EB%8A%A5%ED%95%A0%EA%B9%8C>)

2) CSR로 처리하는 경우에만 프론트에서 백엔드에게 로그인한 사용자 세션을 유지하면서 보낼 수 있는 이유는 다음과 같다고 한다…

“SvelteKit 앱이 export const csr=true로 설정되어 있는 경우 앱은 클라이언트 측 렌더링을 사용합니다. 즉, 세션 ID 값은 브라우저에서 그대로 백엔드로 전송됩니다. 단, export const csr=false로 설정하면 서버 측 렌더링이 사용되며 SvelteKit은 브라우저에서 받은 세션 ID 값이 아닌 백엔드로 전송되는 새로운 세션 ID 값을 생성합니다.  
  
이 동작은 서버 측 렌더링 중 SvelteKit 애플리케이션이 서버 상에서 실행되고 사용자의 세션 상태를 유지하기 위해 서버 상에서 세션 ID를 생성해야 하기 때문에 예상됩니다. 클라이언트 측 렌더링을 사용할 경우 브라우저는 세션 ID를 생성하고 서버는 세션 ID를 검증하기만 하면 됩니다.  
  
코드 내 fetch 메서드에 대해서는 credentials 옵션이 'include'로 설정되어 있습니다. 이는 세션 ID를 포함한 cookie가 요청에 포함된다는 것을 의미합니다. 이 옵션은 요청 중에 사용자의 세션 상태를 유지하기 위해 필요합니다.”

When the SvelteKit app is set to **export const csr = true**, the app will use client-side rendering, which means that the session ID value will be received from the browser and sent to the backend as is. However, when the app is set to **export const csr = false**, it will use server-side rendering, and SvelteKit will generate a new session ID value that is sent to the backend instead of the one received from the browser.

This behavior is expected because, during server-side rendering, the SvelteKit app is running on the server, and the session ID needs to be generated on the server to maintain the user's session state. When using client-side rendering, the browser generates the session ID, and the server only needs to validate it.

Regarding the **fetch** method in your code, the **credentials** option is set to **'include'**, which means that cookies, including the session ID, will be included in the request. This option is necessary to maintain the user's session state during the request.

In summary, the behavior you described is expected when using server-side rendering in SvelteKit, and the **credentials** option in the **fetch** method is necessary to maintain the user's session state during the request.