OAuth 2를 이용한 SSO 환경 구축

넥스트리소프트㈜

목차

1. 개요

한 회사에서 여러 시스템을 운영 중이고, 각 시스템마다 계정을 별도로 관리하여 서비스하는 경우가 있습니다. 시스템들을 이용하는 사용자는 자신이 사용하는 시스템들의 계정을 모두 기억해야 하고, 시스템들을 사용하기 위해서 각 시스템에 별도로 로그인해야 하는 불편함이 있습니다. 또한, 시스템을 운영하는 관리자 역시 회원 정보가 흩어져 있어 회원 관리의 어려움이 증대하게 됩니다.

SSO(Single Sign On)를 도입하면 이러한 문제들을 해결할 수 있습니다. 사용자는 계정 하나만 기억하면 되고, 여러 시스템 사용 시 한번만 로그인하면 됩니다. 관리자 역시 회원 정보를 한 곳에서 관리할 수 있어서 관리 포인트를 줄일 수 있습니다.

SSO를 구성하는 방안은 여러 가지가 있지만 크게 두 가지로 요약할 수 있습니다. 첫째는 예전에 많이 선택한 방법으로 SSO 전용 오픈 소스인 CAS 서버를 이용하는 방안이고, 나머지 하나는 서비스 자원에 대한 접근 제어 인가 흐름(Authorization Flow)을 제공하는 OAuth 2를 이용하여 SSO를 구성하는 것입니다. OAuth 2를 이용하면 웹, 모바일, 데스크탑, IoT 장치로의 확장이 뛰어납니다.

이 글에서는 OAuth 2를 이용해서 SSO 환경을 구성하는 방법에 대해 설명하겠습니다. OAuth 2를 먼저 간략히 살펴보고 SSO 환경을 구성하기 위해 어떻게 OAuth 2를 확장하는지 알아보겠습니다. 마지막으로 스프링부트와 스프링 시큐리티 OAuth로 SSO 환경 구축에 대해 설명하겠습니다.

2. OAuth 2

API를 통해 특정 시스템의 보호된 자원에 접근하기 위해서는 해당 시스템의 사용자 인증 정보(아이디, 패스워드)를 알고 있어야 합니다. 시스템 차원에서 다른 시스템의 보호된 자원에 접근하기 위해 그 시스템의 사용자 인증 정보를 관리하고 이 정보를 사용하는 것은 보안상 많은 문제들을 유발할 수 있습니다. 이런 문제들이 발생하지 않도록 API 접근을 위임하여 인증을 처리하는 방법을 사용할 수 있습니다.

많은 서비스 제공자(Google, Yahoo, AOL 등)들이 이런 인증 방식을 별도로 구현하여 사용하였습니다. 이렇게 구현된 결과들은 서로 조금씩 다르고 서로 호환되지 않았는데 이를 통일하기 위해 OAuth 1.0 표준안이 만들어졌습니다.

OAuth 1.0 발표 이후 몇 년 동안 사용하면서 커뮤니티는 아래와 같은 단점들에 대해 고민하였고 그 결과 OAuth 2(이후 'OAuth'로 명명)를 표준으로 정리하였습니다.

- 암호화 요구사항의 복잡성

- 웹 기반 애플리케이션만 지원
- 성능 확장에 어려움

2.1. OAuth 역할들(Roles)

OAuth에서 위임 방식의 인가를 처리하기 위해 4 개의 역할들을 정의하고 있으며 이들은 아래 표와 같습니다.

역할	설명
자원 서버 (Resource Server)	자원 서버는 보호된 정보를 제공하는 서버로 일반적으로 웹 애플리케이션입니다.
자원 소유자 (Resource Owner)	자원 소유자는 자원 서버에 계정을 가지고 있는 사용자로 클라이언트가 그들의 계정을 통해 자원 서버에 접근하는 것을 인가(authorize)합니다.
클라이언트 (Client)	클라이언트는 자원 소유자를 대신하여 자원 서버의 서비스를 사용하고자 하는 애플리케이션입니다.
인가 서버 (Authorization Server)	인가 서버는 클라이언트가 자원 서버의 서비스를 사용할 때 사용하는 접근 토큰(Access Token)을 발행합니다.

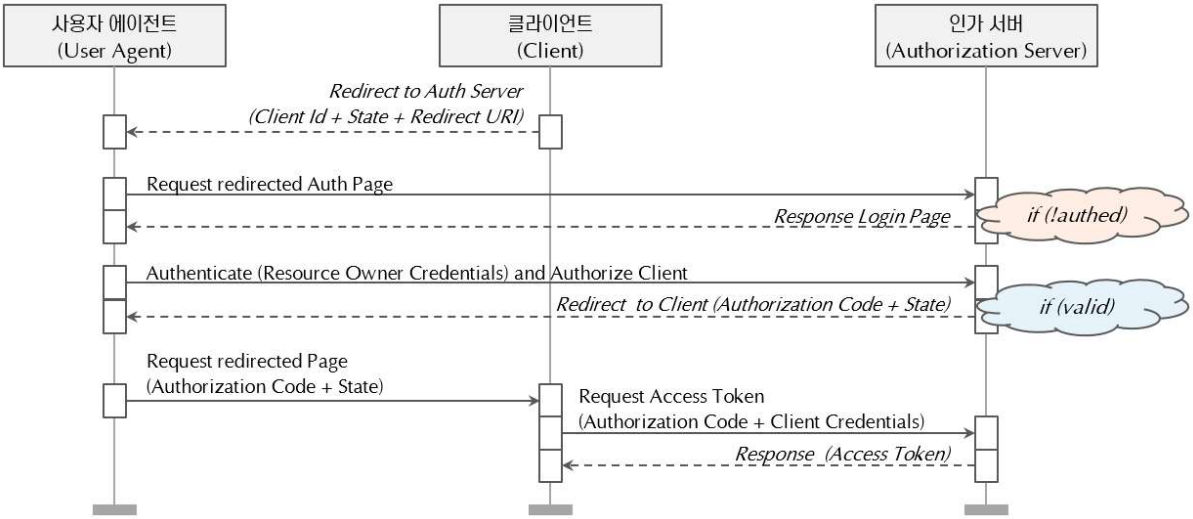
2.2. 인가 승인 유형(Authorization Grant Types)

클라이언트가 접근 토큰을 요청하기 위해서는 자원 소유자의 인가를 받아야 하는데 OAuth는 서로 다른 용도를 위해 다양한 인가 승인 유형을 제공합니다. OAuth에서 정의한 4가지 인가 승인 유형은 아래와 같습니다.

인가 승인 유형	용도	비고
인가 코드 승인 (Authorization Code)	웹 서버 상에서 동작하는 애플리케이션	가장 많이 사용되는 유형
암시적 승인 (Implicit)	모바일 앱 또는 단말기에서 동작하는 웹 애플리케이션	
자원 소유자 패스워드 승인 (Resource Owner Password)	단말기 OS 또는 높은 신뢰 관계의 애플리케이션	다른 유형들을 사용할 수 없는 경우에만 사용
클라이언트 인증 정보 승인 (Client Credentials)	애플리케이션 API 접근	신뢰하는 클라이언트만 사용

2.2.1. 인가 코드 승인(Authorization Code Grant)

인가 코드 승인 방식은 HTTP 리다이렉션 기반 흐름이어서 클라이언트는 자원 소유자의 웹 브라우저와 상호작용할 수 있어야 하며 인가 서버로부터의 요청을 처리할 수 있어야 합니다. 승인 흐름은 아래 그림과 같습니다.



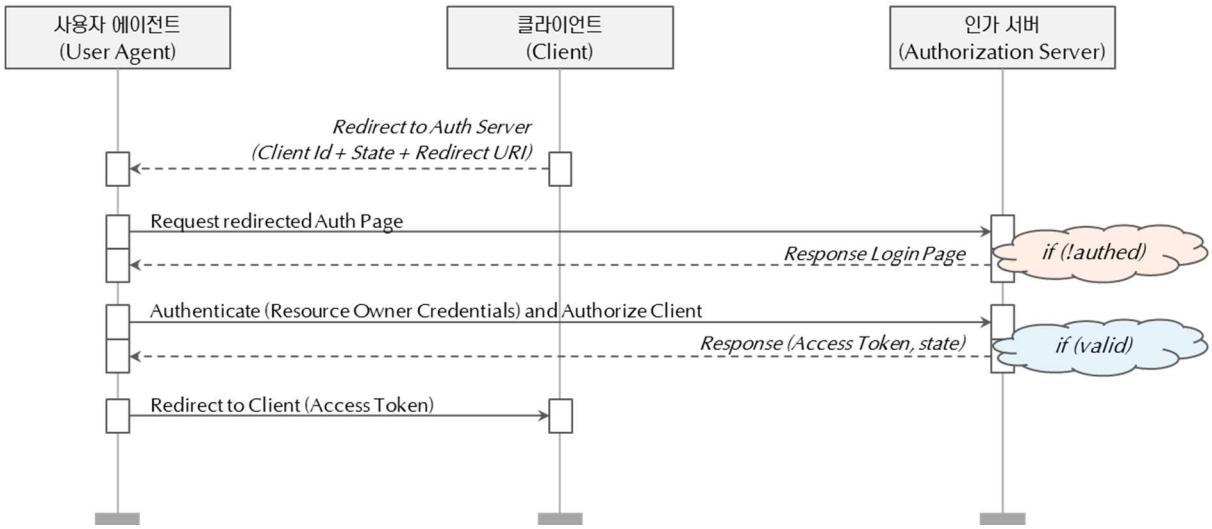
1. 클라이언트가 사용자 에이전트를 인가 서버로 리다이렉트 함으로써 흐름을 시작합니다. (상태(State) 값과 인가 서버로부터 응답을 수신할 Redirect URI 포함)
2. 유효한 인증 이력이 없는 인증 요청에 대해 인가 서버는 로그인 페이지로 응답합니다.
3. 자원 소유자는 인가 서버의 인증을 통해 클라이언트를 인가합니다.
4. 인가 서버는 정상적인 자원 소유자의 인증/인가 요청을 클라이언트로 리다이렉트 합니다. (인가 코드, 상태 값 포함)
5. 인가 코드를 포함한 요청을 받은 클라이언트는 인가 서버로 접근 토큰 요청을 전송하여 그 결과로 접근 토큰을 획득합니다. (인가 코드, 클라이언트 인증 정보 포함)

이 방식은 위 그림에서 볼 수 있듯이 클라이언트의 인증 과정을 거치고 인가 서버와 클라이언트 간에만 접근 토큰이 포함된 통신을 수행하는 등 보안 상 많은 이점을 포함하고 있습니다.

2.2.2. 암시적 승인(Implicit Grant)

암시적 승인은 브라우저에서 자바스크립트와 같은 스크립트 언어로 동작하는 클라이언트들을 지원하기 위한 승인 유형입니다. 웹 브라우저의 신뢰도가 높고, 신뢰할 수 없는 사용자나 애플리케이션에 노출될 염려가 적을 때 사용합니다. 모바일 앱 또는 단말기에서 동작하는 웹 애플리케이

션에서 주로 사용됩니다. 승인 흐름은 아래 그림과 같습니다.

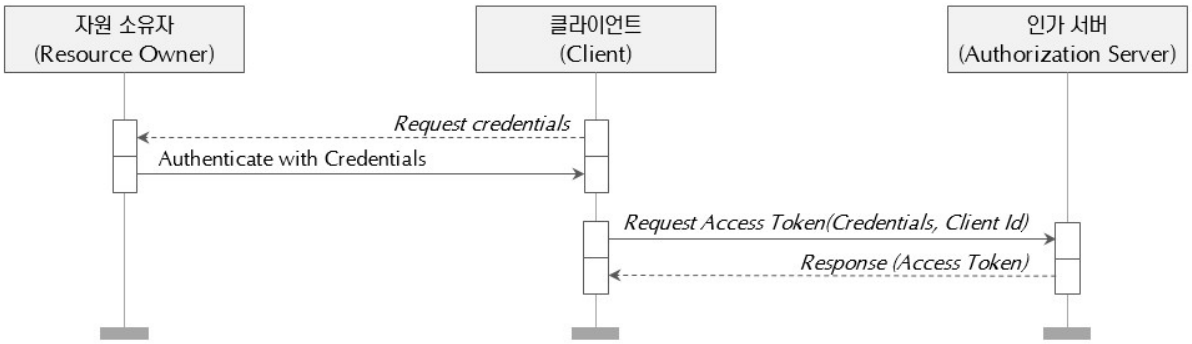


1. 클라이언트가 사용자 에이전트를 인가 서버로 리다이렉트 함으로써 흐름을 시작합니다. (상태(State) 값과 인가 서버로부터 응답을 수신할 Redirect URI 포함)
2. 유효한 인증 이력이 없는 인증 요청에 대해 인가 서버는 로그인 페이지로 응답합니다.
3. 자원 소유자는 인가 서버의 인증을 통해 클라이언트를 인가합니다.
4. 인가 서버는 정상적인 자원 소유자의 인증/인가 요청에 접근 토큰으로 응답합니다.
5. 클라이언트는 접근 토큰을 획득합니다.

이 방식은 재발급 토큰을 사용할 수 없기 때문에 접근 토큰이 만료되면 클라이언트는 접근이 필요한 경우 승인 흐름을 다시 진행해야 합니다.

2.2.3. 자원 소유자 패스워드 승인(Resource Owner Password Grant)

자원 소유자 패스워드 승인은 사용자 이름과 비밀번호를 접근 토큰으로 교환합니다. 승인 흐름은 아래 그림과 같습니다.



1. 클라이언트는 사용자에게 사용자 인증 정보를 요청합니다.
2. 입력 받은 사용자 인증 정보와 클라이언트 정보를 포함하여 인가 서버로 접근 토큰 요청을 보냅니다.
3. 인가 서버는 정상적인 자원 소유자의 인증/인가 요청에 대해 접근 토큰으로 응답합니다.

자원 소유자의 비밀번호가 클라이언트에 노출되기 때문에 다른 유형의 승인 방식을 사용할 수 없는 경우에만 사용해야 합니다.

2.2.4. 클라이언트 인증 정보 승인(Client Credentials Grant)

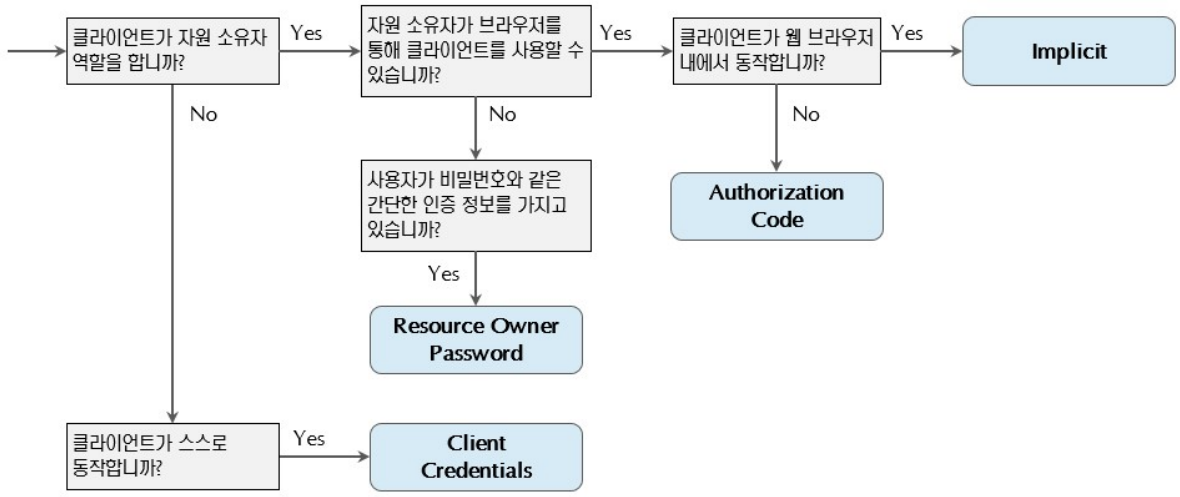
클라이언트 인증 정보 승인은 클라이언트가 데이터를 소유하고 있어서 자원 소유자에게 접근을 위임 받을 필요가 없거나, 외부에서 애플리케이션에 접근 위임이 이미 허용되었을 경우에 사용합니다. 사용자와 관계없는 애플리케이션 API 접근 시에 주로 사용됩니다. 승인 흐름은 아래 그림과 같습니다.

1. 클라이언트는 클라이언트 인증 정보를 담은 요청을 인가 서버로 보냅니다.

2. 인가 서버는 정상적인 클라이언트 인증/인가 요청에 대해 접근 토큰으로 응답합니다.

이 방식은 정확한 사용 사례에 따라 사용되어야 합니다. 고도의 비밀 유지가 필요한 클라이언트를 인증하는 것은 매우 위험하므로, 이런 경우에는 인증 정보를 규칙적으로 바꿔줘야 합니다.

2.3. 인가 승인 유형 선택



클라이언트가 특정 자원소유자를 대신하여 동작하고, 사용자가 웹 브라우저를 사용한다면 리다이렉트 기반 승인방식(인가 코드 승인/ 암시적 승인)을 사용할 수 있습니다. 이 두 가지 중 하나의 선택은 클라이언트 상황에 의존합니다. 클라이언트가 브라우저 자체에서 생명주기를 가진다면 암시적 승인 방식을 선택하면 좋고, 그렇지 않다면 가장 안전하고 유연한 인가 코드 승인 방식을 사용하는 것이 좋습니다.

클라이언트가 사용자와 관계없이 스스로 동작한다면 클라이언트 인증 정보 승인 방식을 선택할 수 있습니다.

사용자가 클라이언트에 공개할 수 있는 간단한 사용자 인증 정보가 존재하고 다른 선택 가능한 방식이 없는 경우에는 자원 소유자 패스워드 승인 방식을 사용할 수 있습니다.

2.4. OAuth 구현 시 보안 고려 사항

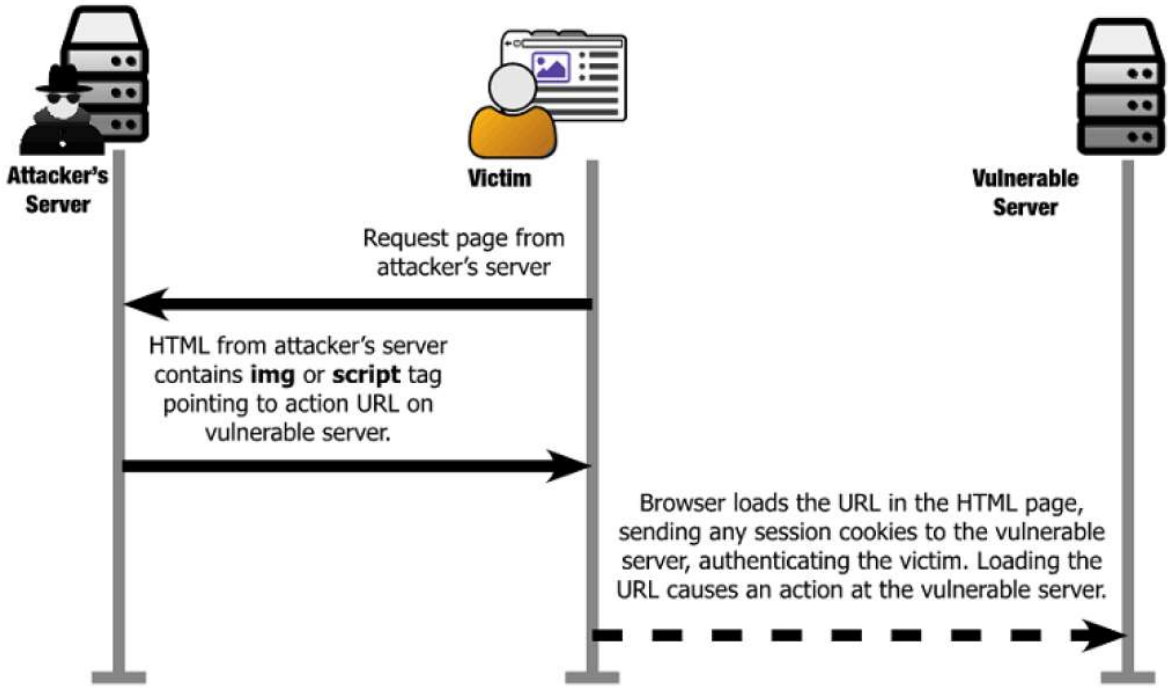
OAuth를 구현할 때 3가지의 시스템, 즉 클라이언트, 인가 서버, 자원 서버를 구현할 수 있습니다. 각각의 역할이 다르며 그에 따른 보안에 대한 고려사항 또한 다르게 적용되어야 합니다. 그리고 최종적으로 발급되는 접근 토큰에 대한 보안도 고려해야 할 사항입니다.

2.4.1. 클라이언트 구현

클라이언트를 구현할 때 중점을 둬야 할 보안 고려사항은 다음과 같습니다.

1) CSRF(cross-site request forgery) 공격

CSRF는 이미 널리 알려지고 흔하게 마주치는 공격입니다. CSRF는 악성 응용 프로그램이 사용자의 브라우저가 인증된 웹 사이트에 대한 요청을 통해 원치 않는 작업을 수행 하게 되는 것입니다.



이에 대한 해결책으로 가장 보편적이며 효과적인 방법은 각 HTTP 요청에 예측할 수 없는 요소를 추가하는 것입니다. CSRF를 피하기 위해 'state' 파라미터의 사용이 권장됩니다. OAuth 클라이언트는 예측할 수 없는 'state' 파라미터 값을 생성하고 인가 서버에 인가 코드 요청과 함께 전달합니다. 인가 서버는 이 값을 그대로 리다이렉트 URI에 대한 파라미터 중 하나로 리턴 합니다. 리턴된 'state' 파라미터를 클라이언트는 자신이 보낸 'state' 파라미터 값과 동일한 지를 확인합니다. 값이 없거나 원래 전달된 값과 일치하지 않으면 클라이언트는 오류를 발생하며 인가 흐름을 종료할 수 있습니다.

2) 클라이언트 정보 공격

암시적 승인(Implicit Grant) 유형을 선택한 클라이언트는 일반적으로 자바 스크립트 전용 애플리케이션인 경우가 많습니다. 브라우저에서 실행중인 클라이언트가 Client Secret을 숨길 수 있는 기능은 제한적입니다. 또한 Client Secret은 컴파일된 코드로 존재해도 안전하지 않습니다. 어떤 코드도 디컴파일될 수 있으며 그 후에는 Client Secret가 안전하지 않기 때문입니다. 이에 대한 방안으로 동적 클라이언트 등록을 사용하여 런타임 시 Client Secret을 구성할 수 있습니다. 인가 서버의

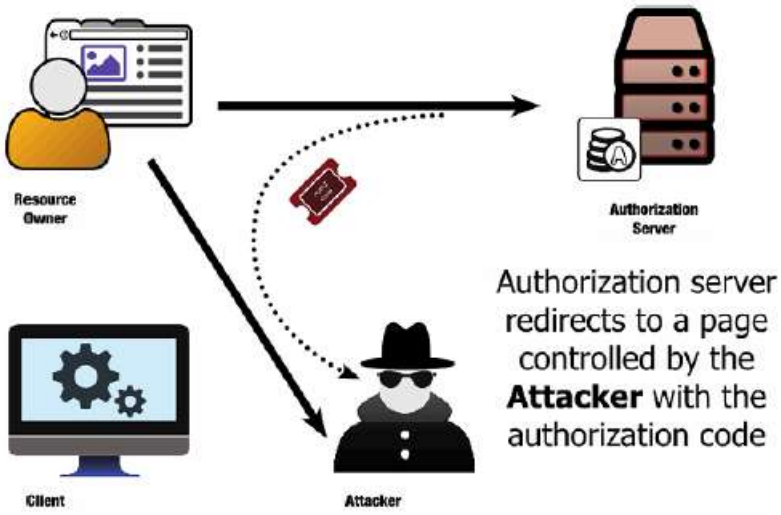
Registration Endpoint를 사용하는 방법입니다. 최초 실행 시에는 Client Id와 Secret이 존재하지 않고 일반적인 OAuth 흐름이 시작되면 Client Id와 Secret은 새로 생성되며 클라이언트 응용 프로그램의 모든 인스턴스에 대해 동일하지 않은 클라이언트 정보로 존재합니다. 클라이언트 애플리케이션의 두 인스턴스는 서로의 정보에 접근할 수 없으며 인가 서버는 인스턴스를 구별할 수 있습니다.

3) 리다이렉트 URI 등록

클라이언트의 정보들을 인가 서버에 등록할 때 리다이렉트 URI 정보가 포함되는데, 이 리다이렉트 URI는 가능한 명확한 URI로 등록하는 것이 중요합니다. 인가 서버는 등록된 리다이렉트 URI의 하위 디렉토리에 대해서도 유연성을 가지고 유효성 검사를 하기 때문입니다. 이 등록에 주의하지 않으면 느슨한 리다이렉트 URI로 인해 공격자로부터 토큰을 도난 당하기 쉬워집니다.

3-1) Referrer를 이용한 인가 코드 도난

클라이언트가 인가 서버에 인가 코드를 요청할 때 요청 파라미터로 인가 코드를 받을 리다이렉트 URI를 넘겨주게 됩니다. 이 때 공격자는 리다이렉트 URI의 하위 디렉토리를 이용하여 HTTP Referrer를 통해 인가 코드를 가로챌 수 있습니다. HTTP Referrer는 한 페이지에서 다른 페이지로 이동할 때 브라우저(일반적으로 HTTP 클라이언트)가 첨부하는 HTTP 헤더 필드입니다. 이것을 이용하여 새 웹 페이지는 원격 사이트의 링크와 같은 요청이 어디에서 왔는지 확인할 수 있습니다.

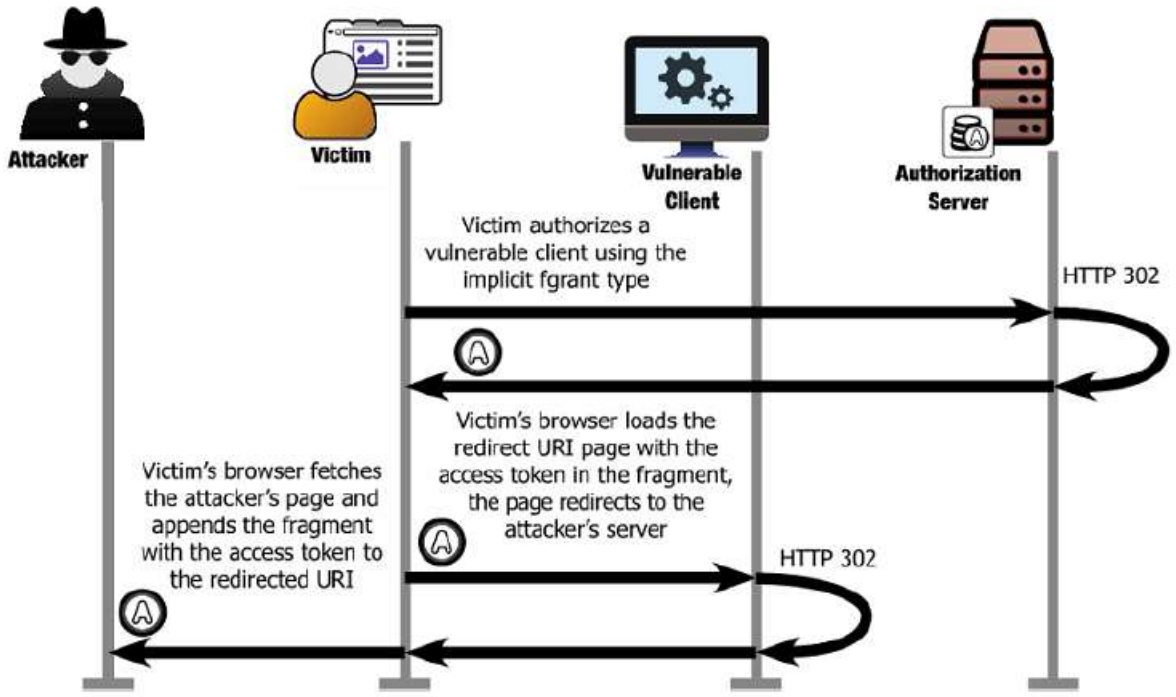


공격자는 유효한 클라이언트의 리다이렉트 URI의 하위 디렉토리에 공격자의 페이지를 만들고, 리다이렉트 URI를 공격자의 페이지로 파라미터를 생성하여 요청을 보냅니다. Referrer에서 인가 코드를 추출하는 것은 공격자의 페이지에 포함된 img 태그에 대한 HTTP 요청을 통해 전달되므로 공

격자에게는 간단한 방법입니다.

3-2) open redirector를 통한 접근 토큰 도난

인가 승인 유형이 암시적 승인(Implicit) 방식인 경우, 공격자의 공격은 인가 코드가 아닌 접근 토큰을 대상으로 합니다. 이 공격은 오픈 리다이렉션이라는 또 다른 일반적인 웹 취약점을 기반으로 합니다. 대부분의 브라우저는 리다이렉션 시 원래의 URI 조각을 보존합니다. 또한 그 조각은 브라우저 내에서만 사용되기 때문에 서버로 전송되지 않습니다. 자원 소유자가 이미 애플리케이션을 승인 한 경우 또는 애플리케이션을 다시 승인하도록 할 수 있는 경우, 자원 소유자는 URI 조각에 접근 토큰이 추가된 리다이렉트 URI로 리다이렉션 됩니다. 이 시점에 리다이렉트 URI가 공격자가 생성한 유효한 리다이렉트 URI의 하위 디렉토리 URI일 경우 공격자의 페이지로 리다이렉션 되고, 공격자는 접근 토큰을 가로챌 수 있게 됩니다.



4) 인가 코드 도난

인가 코드를 사용해서 접근 토큰을 발급 받으려면 Client Secret이 필요하며 이는 안전하게 보호되어야 합니다. 그러나 클라이언트가 공용 클라이언트인 경우에는 Client Secret이 없으므로 인가 코드를 공격자가 가로채기가 쉽습니다. 기밀 클라이언트의 경우 공격자는 CSRF를 통해 Client Secret을 악의적으로 얻으려고 시도할 수 있으므로 주의해야 합니다.

5) 접근 토큰 도난

공격자의 궁극적인 목표는 접근 토큰을 가로채는 것입니다. 접근 토큰을 사용하면 정상적으로 수행할 수 없는 모든 종류의 작업을 공격자가 수행할 수 있게 됩니다. OAuth 스펙에서 Bearer 토큰을 전달하는 방법 중 하나로 클라이언트가 쿼리 파라미터를 이용하여 접근 토큰을 URI로 보내는 방법이 존재합니다. 이 방법이 단순함으로 인해 유혹을 불러 일으키지만 접근 토큰을 자원 서버로 제출할 때 이 방법을 사용하면 많은 단점이 있습니다. 먼저, 접근 토큰은 URI의 일부로 access.log 파일에 기록됩니다. 또한 Referrer에는 URL이 포함되기 때문에 Referrer를 통한 접근 토큰 누출의 위험이 있습니다. 쿼리 파라미터는 여전히 OAuth에 유효한 방법이지만 클라이언트는 최후의 수단으로 극도의 주의를 기울여 이 방법을 사용해야 합니다.

2.4.2. 인가 서버 구현

인가 서버를 구현할 때 중점을 둬야 할 보안 고려사항은 다음과 같습니다.

1) 세션 Hijacking

인가 코드는 쿼리 파라미터로 넘어오므로 브라우저 기록에 유지됩니다. 로그아웃을 해도 브라우저 기록은 지워지지 않습니다. 공격자가 동일한 브라우저를 이용하여 자신의 정보로 로그인을 하고 브라우저 기록에 있는 이전 자원 소유자 세션의 인가 코드를 삽입하면 이전 자원 소유자의 자원에 접근할 수 있게 됩니다. 그렇기 때문에 인가 서버는 인가 코드를 두 번 이상 사용하면 요청을 거부해야 하며 가능하면 인가 코드를 기반으로 이전에 발행된 모든 토큰을 취소하게 구현해야 합니다. 이러한 검사가 없으면 다른 클라이언트에 대해 발행된 인가 코드를 사용하여 접근 토큰을 얻을 수 있기 때문입니다.

2) 리다이렉트 URI 유효성 검사

클라이언트 구현 시 리다이렉트 URI 등록의 정확성이 얼마나 중요한 지 살펴보았는데, 이와 마찬가지로 인가 서버에서 리다이렉트 URI를 검사하는 것도 중요한 보안 고려사항입니다. 인가 서버에 등록된 리다이렉트 URI와 쿼리 파라미터로 넘어온 URI의 유효성을 검사해야 합니다. 리다이렉트 URI 유효성 확인은 정확히 일치하는지 검사하는 방법, 하위 디렉토리까지 허용하는 방법 그리고, 서브 도메인까지 허용하는 방법이 일반적으로 많이 사용됩니다. 너무 느슨한 리다이렉트 URI 유효성 검사는 공격자로부터 접근 토큰을 가로채기 쉽게 만들기 때문에 가급적이면 정확히 일치하는지 검사하는 방법의 사용을 권합니다.

3) 클라이언트 유효성 검사

인가 서버는 등록된 클라이언트와 요청된 클라이언트가 동일한 클라이언트인지 판단하는 것이 중요합니다. 상태(State) 값이나 Client 식별자 정보처럼 리다이렉트 URI에 포함된 파라미터가 초기 승인 요청에 포함되어 있는지 확인하고, 해당 값이 동일한지 확인하는 작업이 필요합니다.

4) Scope 값의 오류

클라이언트로부터 요청된 리다이렉트 URI 중 'scope' 값이 올바르지 않으면 인가 서버는 클라이언트로 오류와 함께 리다이렉트 합니다. 이를 이용하여 공격자는 접근 토큰을 가로챌 수도 있습니다. 오류와 함께 리다이렉트된 URI를 이용하여 실제 유효한 클라이언트 요청에 이를 끼워 넣어 접근 토큰을 가로채는 방식입니다. 그러므로 클라이언트로부터 scope의 값이 존재하지 않거나 틀린 값으로 요청된 경우에는 리다이렉트 URI로 리다이렉션하지 않고 400 오류를 발생시키는 것이 최선의 방법입니다.

2.4.3. 자원 서버 구현

자원 서버 또한 보안에 최적화 되도록 구현해야 합니다.

1) 자원 Endpoint의 설계

일부 사용자의 입력에 의해 결과 응답이 구동되는 REST API를 설계하는 경우 XSS 취약성이 발생할 위험이 높습니다. 현대 브라우저에서 제공하는 기능과 웹에서 리소스가 노출되는 일반적인 시점에 대한 모범 사례를 가능한 한 많이 활용해야 합니다.

XSS를 막는 첫 번째 방법으로는 쿼리 파라미터에 대해 이스케이프 처리를 하는 것입니다. 공격자가 악의적으로 쿼리 파라미터에 스크립트 태그를 추가하여 동작하도록 할 수 있기 때문에 이스케이프 처리를 하여 태그를 인식할 수 없도록 구현할 수 있습니다.

두 번째 방법으로는 알맞은 Content-type을 반환하도록 하여 XXS를 막을 수도 있습니다. 이 또한 공격자가 작성한 스크립트 태그를 방지하기 위해 사용할 수 있는 방법입니다.

또한 자원 서버에서 접근 토큰을 쿼리 파라미터로 전달받지 않도록 할 수 있습니다. 이렇게 하면 공격자가 접근 토큰을 포함하는 URI를 위조할 수 있는 방법은 없습니다.

2) 동일 출처 정책 문제

웹에서는 클라이언트 응용 프로그램이 한 도메인에서 제공되는 반면 보호된 자원은 다른 도메인에서 제공되는 것이 일반적입니다. 그러나 브라우저는 한 페이지 내의 자바 스크립트가 다른 도메인의 악의적인 콘텐츠를 로드하지 못하도록 동일한 출처 정책이 설정됩니다. 이 경우, API 호출을 자바 스크립트 호출로 허용하는 것이 좋습니다. 특히 OAuth를 사용하여 API를 보호해야 하기

때문에 이 문제를 해결하기 솔루션을 사용하는 방법을 선택해야 할 수도 있습니다.

3) 토큰의 수명

공격자가 접근 토큰을 획득하면 보호된 자원에 언제든지 접근할 수 있게 됩니다. 이러한 위험을 최소화하기 위해 상대적으로 수명이 짧은 접근 토큰을 갖는 것이 중요합니다. 실제로 공격자가 사용자의 접근 토큰을 확보하더라도 이미 만료되었거나 만료일에 가까워지면 공격의 심각도는 감소합니다. 이러한 이유로 OAuth 시스템 전체에서 가능한 한 TLS(Transport Layer Security) 사용을 강제하는 것이 가장 좋습니다. 그리고 브라우저 보호 및 보안 헤더를 최대한 활용하는 것도 접근 토큰을 공격자로부터 지키는 방법입니다.

2.4.4. 접근 토큰 관련

OAuth 스펙은 bearer 토큰을 보안 장치로 정의합니다. 이 토큰을 소유한 당사자는 해당 당사자가 누구인지에 관계없이 토큰을 사용할 수 있습니다. 그러므로 클라이언트에 대한 접근권한이 부여되며 누가 사용하는지 상관하지 않습니다.

1) Bearer 토큰 사용의 위험 및 고려 사항

접근 토큰이 위조될 경우가 발생합니다. 공격자는 자체 위조 토큰을 만들거나 유효한 토큰을 수정하여 자원 서버가 클라이언트에 부적절한 접근을 허용하게 할 수 있습니다. 또는 공격자가 토큰 자체의 유효성을 확장할 수 있습니다.

공격자는 이미 사용된 이전 토큰을 사용하려고 시도할 것입니다. 이 경우 자원 서버는 유효한 데이터를 반환하지 않아야 합니다. 대신 오류를 리턴 해야 합니다.

공격자는 토큰 리다이렉션을 이용하는 경우도 있습니다. 공격자는 한 자원 서버에서 사용하도록 발급된 토큰을 사용하여 토큰이 유효하다고 생각되는 다른 자원 서버에 접근합니다. 이 경우 공격자가 특정 자원 서버에 대한 접근 토큰을 합법적으로 가져 와서 이 접근 토큰을 다른 리소스 토큰에 표시하려고 할 것입니다.

2) bearer 토큰 전송 보호

접근 토큰이 안전하지 않은 채널을 통해 전송되는 것을 조심해야 합니다. OAuth 스펙에 명시된 것처럼 접근 토큰의 전송은 네트워크의 통신 보안을 제공하도록 설계된 암호화 프로토콜인 SSL / TLS를 사용하여 보호되어야 합니다.

3) 클라이언트에서 주의해야 할 사항

클라이언트가 적용할 수 있는 하나의 대책은 토큰의 범위를 작업에 필요한 최소값으로 제한하

는 것입니다. 자원 소유자의 최소한의 자원을 묻는 것만으로 충분하다면 최소한의 권한을 주는 것이 중요합니다. 또한 가능하다면 일시적인 메모리에 접근 토큰을 유지하는 것이 좋습니다. 데이터베이스나 다른 저장소에 토큰을 저장한다면 공격자가 데이터베이스나 저장소의 권한 획득으로 토큰이 노출될 위험이 있기 때문입니다.

4) 인가 서버에서 주의해야 할 사항

대부분의 경우 인가 서버는 접근 토큰을 데이터베이스에 저장합니다. 그러므로 공격자가 인가 서버 데이터베이스에 접근하거나 SQL 인젝션을 할 수 있으면 여러 자원 소유자의 보안이 손상될 수 있습니다. 인가 서버는 토큰 자체의 텍스트 대신 접근 토큰의 해시 (예: SHA-256)를 저장하는 방식을 택할 수 있습니다. 이 경우 공격자가 모든 접근 토큰을 포함하는 전체 데이터베이스를 도용할 수 있더라도 유출된 정보로 수행할 수 있는 일은 많지 않습니다.

또한 단일 접근 토큰의 누출과 관련된 위험을 최소화하려면 접근 토큰의 수명을 짧게 유지하는 것이 좋습니다. 클라이언트가 자원에 더 오래 접근해야 하는 경우 인가 서버는 클라이언트에 Refresh 토큰을 발행할 수 있습니다.

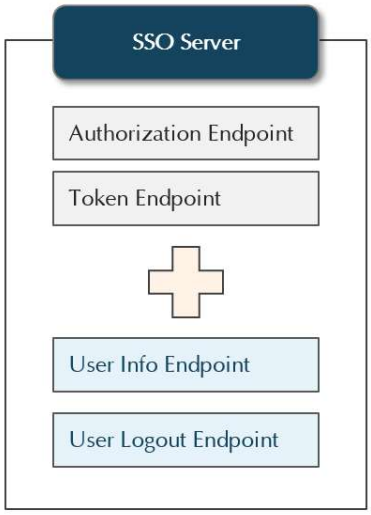
궁극적으로 인가 서버에서 수행할 수 있는 가장 좋은 작업 중 하나는 보안 로깅입니다. 토큰이 발행, 사용 또는 취소 될 때마다 보안 로그를 기록하면 의심되는 동작을 감시하는 데 사용할 수 있습니다. 그리고 이러한 모든 로그는 노출되지 않도록 지켜져야 합니다.

5) 자원 서버에서 주의해야 할 사항

자원 서버는 토큰의 유효성을 적절히 검사하고 일종의 수퍼 파워를 가진 특수 목적의 접근 토큰을 사용하지 않아야 합니다.

3. SSO 환경 구성을 위한 OAuth 확장

OAuth를 사용해서 SSO를 구성하기 위해 인가 승인 유형 중 인가 코드 승인 방식을 사용하였고, OAuth가 기본으로 제공하는 2개의 Endpoint와 별도 구현한 2개의 Endpoint로 SSO 서버를 구성하였습니다.

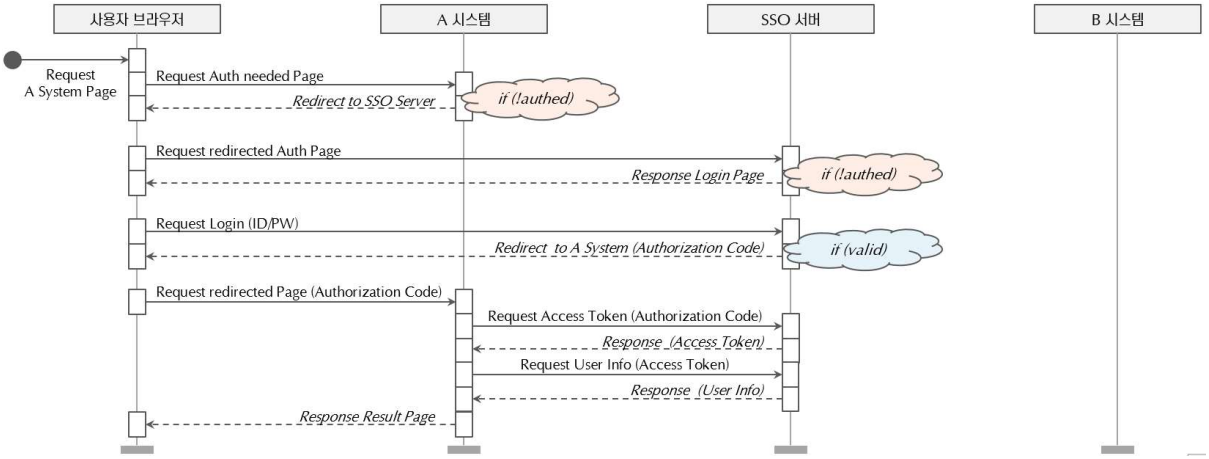


채널 구분	URI	용도
Front Channel	/authorize	클라이언트 인가
Back Channel	/token	접근 토큰 발급
Back Channel	/userInfo	사용자 정보 조회
Front Channel	/userLogout	로그아웃

위와 같이 구성한 SSO 환경에서의 로그인, SSO 처리, 로그아웃 시나리오를 차례대로 살펴보겠습니다.

3.1. 로그인 시나리오

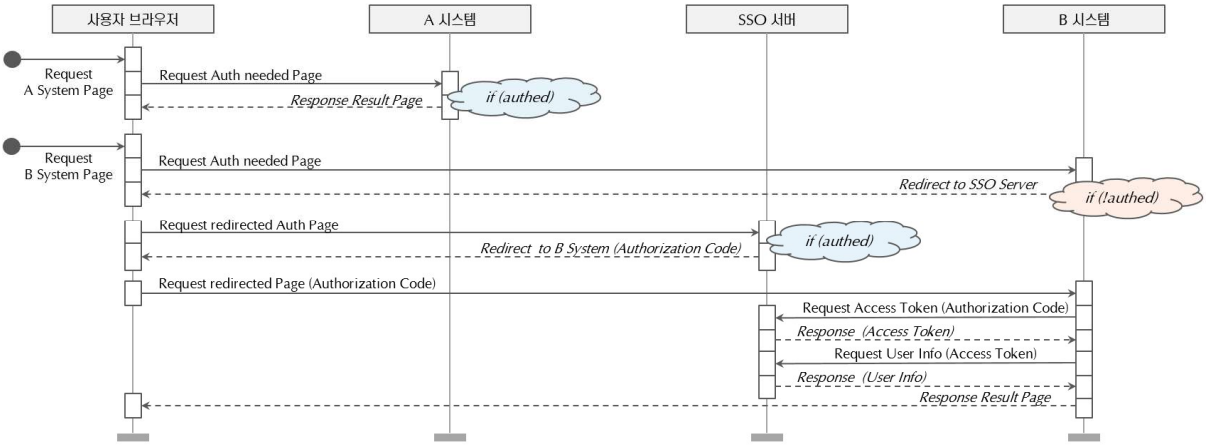
구성한 SSO 환경의 로그인 시나리오는 아래 그림과 같습니다.



1. 각 시스템들은 인증되지 않은 사용자의 요청을 SSO 서버로 리다이렉트합니다.
2. 유효한 인증 이력이 없는 인증 요청에 대해 SSO 서버는 로그인 페이지로 응답합니다.
3. 유효한 로그인 정보로 로그인 요청을 받은 SSO 서버는 이 요청을 시스템으로 리다이렉트 (인가 코드 포함)합니다.
4. 인가 코드를 포함한 요청을 받은 시스템은 SSO 서버로부터 접근 토큰과 사용자 정보를 획득하여 해당 사용자의 로그인 처리를 한 후 원래 요청의 결과로 응답합니다.

3.2. SSO 처리 시나리오

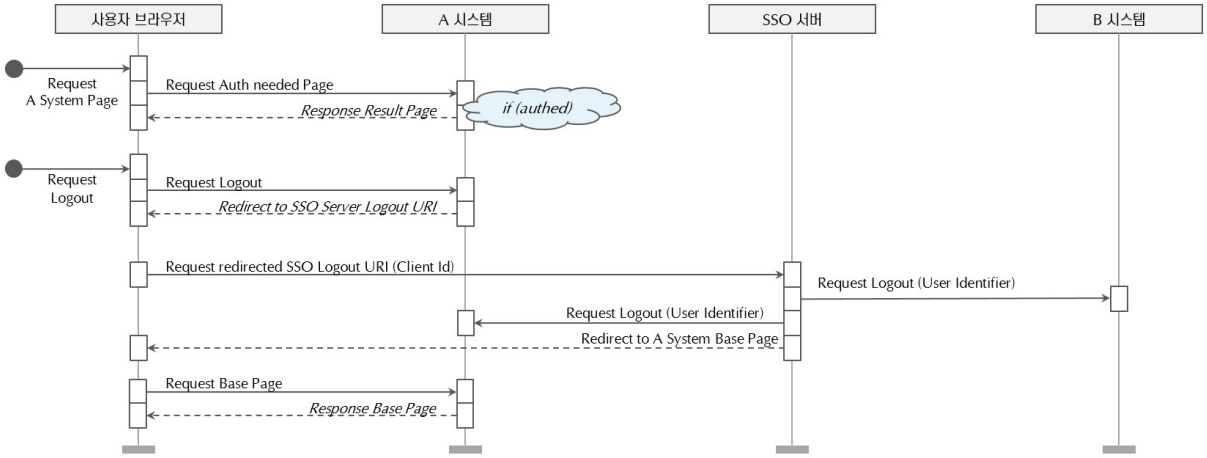
SSO 구성 환경에서 SSO 처리 시나리오는 아래 그림과 같습니다.



1. 한 시스템을 통해 로그인한 사용자가 다른 시스템의 인증이 요구되는 페이지를 요청하면 해당 시스템은 이 요청을 SSO 서버로 리다이렉트 합니다.
2. 유효한 인증 이력이 있는 요청에 대해 SSO 서버는 이 요청을 시스템으로 리다이렉트 (인가 코드 포함)합니다.
3. 인가 코드를 포함한 요청을 받은 시스템은 SSO 서버로부터 접근 토큰과 사용자 정보를 획득하여 해당 사용자의 로그인 처리를 한 후 원래 요청의 결과로 응답합니다.

3.3. 로그아웃 시나리오

로그인한 사용자가 SSO 환경에서 로그아웃 시 처리 과정은 아래 그림과 같습니다.



1. 로그인한 사용자가 한 시스템에게 로그아웃을 요청하면 시스템은 이 요청을 SSO 서버의 로그아웃 페이지로 리다이렉트 합니다.
2. 로그아웃 요청을 받은 SSO 서버는 해당 사용자로 로그인된 모든 시스템에게 로그아웃을 요청합니다.

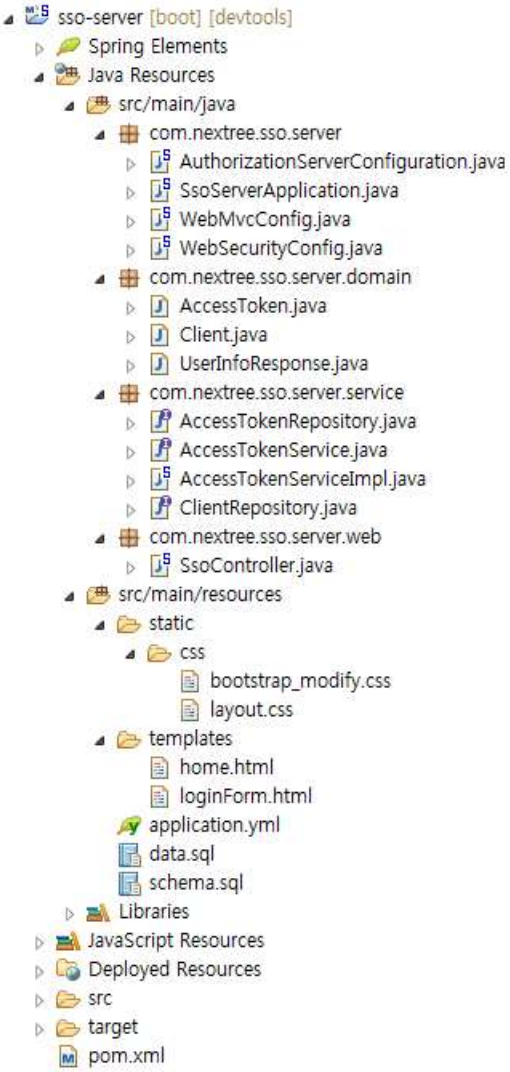
4. SSO 환경 구축

앞에서 설명한 내용으로 SSO 환경을 구축하기 위해 스프링 부트와 스프링 시큐리티 OAuth2를 사용하였습니다. 먼저 SSO 서버 구축에 대해 설명한 후 클라이언트 시스템 구축에 대해 알아보겠습니다.

4.1. SSO 서버 구축

4.1.1. 프로젝트 구성

SSO 서버 프로젝트의 전체 구조는 아래 그림과 같습니다. 프로젝트 구성을 간단히 하기 위해 테스트 관련 사항들은 기술하지 않았습니다.



메이븐(Maven)으로 프로젝트를 생성한 후 스프링 부트와 기타 라이브러리를 사용할 수 있도록 아래와 같이 pom.xml 파일을 편집합니다.

```
<project ……>
  ……
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.2.RELEASE</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.security.oauth</groupId>
      <artifactId>spring-security-oauth2</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-data-jpa</artifactId>
      </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-thymeleaf</artifactId>
      </dependency>
      <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <optional>true</optional>
    </dependency>

    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.httpcomponents</groupId>
      <artifactId>httpclient</artifactId>
    </dependency>

    <dependency>
      <groupId>org.webjars</groupId>
      <artifactId>jquery</artifactId>
      <version>3.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.webjars</groupId>
      <artifactId>bootstrap</artifactId>
      <version>3.3.7</version>
    </dependency>
  </dependencies>
  ……
</project>
```

4.1.2. 스프링 부트 애플리케이션 설정 파일

src/main/resources 폴더 하위에 애플리케이션 설정 파일인 application.yml을 아래와 같이 작성합니다.

```
server:
  port: 8080
  session:
    cookie:
      name: APPSESSIONID

spring.h2.console:
  enabled: true
  path: /h2-console

spring:
  jpa:
    generate-ddl: false
    hibernate:
      ddl-auto: none

logging.level:
  root: warn
  org.springframework:
    web: warn
    security: info
    boot: info
  org.hibernate:
    SQL: warn
  com.nextree: debug
```

애플리케이션이 동작할 톰캣 서버의 포트를 8080으로, 세션 쿠키의 이름을 "APPSESSIONID"로 설정하였습니다. SSO 서버 프로젝트에서 기본 데이터베이스로 H2 데이터베이스를 사용하는데 H2 데이터베이스의 콘솔을 사용하도록 하고 콘솔의 URL 경로를 "/h2-console"로 설정하였습니다. 데이터베이스에 저장된 클라이언트 및 접근 토큰 정보에 접근하기 위해 JPA를 사용며 애플리케이션 구동 시 DDL을 생성 하지 않도록 설정하였습니다.

4.1.3. 스프링 부트 애플리케이션 클래스

아래와 같이 SpringBootApplication 어노테이션을 적용한 클래스를 작성하였습니다. 이 클래스를 구동시키면 SSO 서버가 시작됩니다.

```
@SpringBootApplication
public class SsoServerApplication {
  //
  public static void main(String[] args) {
    //
    SpringApplication.run(SsoServerApplication.class, args);
  }
}
```

4.1.4. 웹 보안 설정 클래스

스프링 시큐리티 설정을 위해 EnableWebSecurity 어노테이션을 적용한 클래스를 아래와 같이 정의하였습니다.

```
@Configuration
@EnableWebSecurity
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
   //
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      //
      http
         .authorizeRequests()
            .antMatchers("/home", "/webjars/**", "/css/**", "/userInfo").permitAll()
            .anyRequest().authenticated()
            .and()
         .formLogin()
            .loginProcessingUrl("/login")
            .loginPage("/loginForm")
            .permitAll()
            .and()
         .csrf()
            .requireCsrfProtectionMatcher(new AntPathRequestMatcher("/user*"))
            .disable()
         .logout()
            .permitAll();
   }

   @Autowired
   public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
      //
      auth
         .inMemoryAuthentication()
         .withUser("tsong").password("aaa").roles("USER").and()
         .withUser("jmpark").password("aaa").roles("USER").and()
         .withUser("jkkang").password("aaa").roles("USER").and()
         .withUser("test").password("aaa").roles("USER");
   }
}
```

configure() 메소드에서 인증이 필요 없는 URI 등록, 폼 로그인 등의 설정을 정의하였고, configureGlobal() 메소드에서는 인증 계정 관리를 메모리 상에서 처리하도록 했으며 테스트용 계정 정보를 설정하였습니다.

4.1.5. 웹 MVC 설정 클래스

서비스 레이어의 처리가 필요 없는 요청(홈 페이지, 로그인 폼 페이지)을 설정 하기 위해 WebMvcConfig 클래스를 아래와 같이 정의하였습니다.

```
@Configuration
public class WebMvcConfig extends WebMvcConfigurerAdapter {
    //
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        //
        registry.addViewController("/home").setViewName("home");
        registry.addViewController("/loginForm").setViewName("loginForm");
    }
}
```

4.1.6. 인가 서버 설정 클래스

스프링 부트로 애플리케이션 작성 시 EnableAuthorizationServer 어노테이션을 적용한 클래스를 정의하면 OAuth 인가 서버로 동작하게 할 수 있습니다.

```
@Configuration
@EnableAuthorizationServer
public class AuthorizationServerConfiguration extends AuthorizationServerConfigurerAdapter {
  //
  @Autowired
  private AuthorizationCodeServices authorizationCodeServices;

  @Autowired
  private ApprovalStore approvalStore;

  @Autowired
  private TokenStore tokenStore;

  @Override
  public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
    //
    endpoints
      .tokenStore(tokenStore)
      .authorizationCodeServices(authorizationCodeServices)
      .approvalStore(approvalStore);
  }

  @Bean
  public AuthorizationCodeServices jdbcAuthorizationCodeServices(DataSource dataSource) {
    //
    return new JdbcAuthorizationCodeServices(dataSource);
  }

  @Bean
  public ApprovalStore jdbcApprovalStore(DataSource dataSource) {
    //
    return new JdbcApprovalStore(dataSource);
  }

  @Bean
  @Primary
  public ClientDetailsService jdbcClientDetailsService(DataSource dataSource) {
    //
    return new JdbcClientDetailsService(dataSource);
  }

  @Bean
  public TokenStore jdbcTokenStore(DataSource dataSource) {
    //
    return new JdbcTokenStore(dataSource);
  }
}
```

SSO 인증 대상 클라이언트, 인가 코드 및 발급한 토큰 정보를 데이터베이스에 저장하기 위해 인가 서버 설정 클래스에 스프링 시큐리티 OAuth2에서 제공하는 JdbcAuthorizationCodeServices, JdbcApprovalStore, JdbcClientDetailsService, JdbcTokenStore를 빈으로 설정하였습니다.

4.1.7. 데이터베이스 스키마 생성 및 초기 데이터 적재 스크립트

인가 서버 클래스 정의에서 등록한 데이터베이스 관련 빈들에서 사용하는 데이터베이스 스키마를 생성하기 위해 src/main/resources 디렉토리에 DDL 문이 포함된 schema.sql 파일을 작성하면 됩

니다. 또한, 초기 데이터 적재가 필요한 경우 src/main/resources 디렉토리에 DML 문이 포함된 data.sql 파일을 작성하면 됩니다. 스프링 부트 애플리케이션 구동 시 이 스크립트들을 실행시켜 스키마를 생성하고 초기 데이터를 적재합니다.

아래는 schema.sql, data.sql 파일 내용입니다.

```
create table oauth_client_details (
    client_id VARCHAR(256) PRIMARY KEY,
    resource_ids VARCHAR(256),
    client_secret VARCHAR(256),
    scope VARCHAR(256),
    authorized_grant_types VARCHAR(256),
    web_server_redirect_uri VARCHAR(256),
    logout_uri VARCHAR(256),
    base_uri VARCHAR(256),
    authorities VARCHAR(256),
    access_token_validity INTEGER,
    refresh_token_validity INTEGER,
    additional_information VARCHAR(4096),
    autoapprove VARCHAR(256)
);

create table oauth_access_token (
    token_id VARCHAR(256),
    token LONGVARBINARY,
    authentication_id VARCHAR(256) PRIMARY KEY,
    user_name VARCHAR(256),
    client_id VARCHAR(256),
    authentication LONGVARBINARY,
    refresh_token VARCHAR(256)
);

create table oauth_refresh_token (
    token_id VARCHAR(256),
    token LONGVARBINARY,
    authentication LONGVARBINARY
);

create table oauth_code (
    code VARCHAR(256), authentication LONGVARBINARY
);

create table oauth_approvals (
    userId VARCHAR(256),
    clientId VARCHAR(256),
    scope VARCHAR(256),
    status VARCHAR(10),
    expiresAt TIMESTAMP,
    lastModifiedAt TIMESTAMP
);
```

```
insert into oauth_client_details (client_id, client_secret, resource_ids, scope, authorized_grant_types,
  web_server_redirect_uri, authorities, access_token_validity, refresh_token_validity, additional_information,
  autoapprove, logout_uri, base_uri) values ('System1_id', 'System1_secret', null, 'read', 'authorization_code',
  'http://localhost:18010/oauthCallback', 'ROLE_YOUR_CLIENT', 36000, 2592000, null, 'true',
  'http://localhost:18010/logout', 'http://localhost:18010/me');
insert into oauth_client_details (client_id, client_secret, resource_ids, scope, authorized_grant_types,
  web_server_redirect_uri, authorities, access_token_validity, refresh_token_validity, additional_information,
  autoapprove, logout_uri, base_uri) values ('System2_id', 'System2_secret', null, 'read', 'authorization_code',
  'http://localhost:18020/oauthCallback', 'ROLE_YOUR_CLIENT', 36000, 2592000, null, 'true',
  'http://localhost:18020/logout', 'http://localhost:18020/me');
insert into oauth_client_details (client_id, client_secret, resource_ids, scope, authorized_grant_types,
  web_server_redirect_uri, authorities, access_token_validity, refresh_token_validity, additional_information,
  autoapprove, logout_uri, base_uri) values ('System3_id', 'System3_secret', null, 'read', 'authorization_code',
  'http://localhost:18030/oauthCallback', 'ROLE_YOUR_CLIENT', 36000, 2592000, null, 'true',
  'http://localhost:18030/logout', 'http://localhost:18030/me');
insert into oauth_client_details (client_id, client_secret, resource_ids, scope, authorized_grant_types,
  web_server_redirect_uri, authorities, access_token_validity, refresh_token_validity, additional_information,
  autoapprove, logout_uri, base_uri) values ('System4_id', 'System4_secret', null, 'read', 'authorization_code',
  'http://localhost:18040/oauthCallback', 'ROLE_YOUR_CLIENT', 36000, 2592000, null, 'true',
  'http://localhost:18040/logout', 'http://localhost:18040/me');
```

4.1.8. 엔터티 클래스

데이터베이스에 저장되어 있는 클라이언트 및 접근 토큰 정보를 조회하기 JPA를 사용하였고 이 때 사용하는 엔터티 클래스들을 아래와 같이 정의하였습니다.

```
@Entity
@Table(name="oauth_client_details")
public class Client {
  //
  @Id
  @Column(name="client_id")
  private String clientId;

  @Column(name="web_server_redirect_uri")
  private String redirectUri;

  @Column(name="logout_uri")
  private String logoutUri;

  @Column(name="base_uri")
  private String baseUri;

  // getter/setter
}
```

```
@Entity
@Table(name="oauth_access_token")
public class AccessToken {
   //
   @Id
   private String tokenId;

   private String token;

   @Column(name="user_name")
   private String userName;

   @Column(name="authentication_id")
   private String authenticationId;

   @Column(name="client_id")
   private String clientId;

   private String authentication;

   // getter /setter
}
```

4.1.9. 데이터베이스 처리를 위한 JPA 리파지토리

Client, AccessToken 엔터티들의 처리를 위해 스프링에서 제공하는 CrudRepository를 사용하는 인터페이스들을 정의하였습니다.

```
public interface ClientRepository extends CrudRepository<Client, String> {
   //
}
```

```
public interface AccessTokenRepository extends CrudRepository<AccessToken, String> {
   //
   AccessToken findByTokenIdAndClientId(String tokenId, String clientId);

   int deleteByUserName(String userName);

   List<AccessToken> findByUserName(String userName);
}
```

4.1.10. 서비스 인터페이스 및 서비스 구현 클래스

접근 토큰 값과 클라이언트 아이디로 AccessToken을 조회하는 메소드와 클라이언트 아이디와 사용자 이름으로 로그아웃을 처리하는 메소드를 정의하는 서비스 인터페이스를 작성하였습니다.

```
public interface SsoService {
   //
   AccessToken getAccessToken(String token, String clientId);

   String logoutAllClients(String clientId, String userName);
}
```

앞에서 정의한 서비스 인터페이스를 구현한 클래스는 아래와 같습니다.

```
@Service("ssoService")
public class SsoServiceImpl implements SsoService {
   //
   private static final Logger log = LoggerFactory.getLogger(SsoServiceImpl.class);

   @Autowired
   private AccessTokenRepository accessTokenRepository;

   @Autowired
   private ClientRepository clientRepository;

   @Override
   public AccessToken getAccessToken(String token, String clientId) {
      //
      String tokenId = extractTokenId(token);

      return accessTokenRepository.findByTokenIdAndClientId(tokenId, clientId);
   }

   private String extractTokenId(String value) {
      //
      if (value == null) {
         return null;
      }

      try {
         MessageDigest digest = MessageDigest.getInstance("MD5");

         byte[] bytes = digest.digest(value.getBytes("UTF-8"));
         return String.format("%032x", new BigInteger(1, bytes));
      }
      catch (NoSuchAlgorithmException e) {
         throw new IllegalStateException("MD5 algorithm not available.  Fatal (should be in the JDK).");
      }
      catch (UnsupportedEncodingException e) {
         throw new IllegalStateException("UTF-8 encoding not available.  Fatal (should be in the JDK).");
      }
   }
```

```
@Override
@Transactional
public String logoutAllClients(String clientId, String userName) {
    //
    requestLogoutToAllClients(userName);

    removeAccessTokens(userName);

    Client client = clientRepository.findOne(clientId);

    return client.getBaseUri();
}

private void requestLogoutToAllClients(String userName) {
    //
    List<AccessToken> tokens = accessTokenRepository.findByUserName(userName);

    for (AccessToken token : tokens) {
        requestLogoutToClient(token);
    }
}

private void requestLogoutToClient(AccessToken token) {
    //
    Client client = clientRepository.findOne(token.getClientId());

    String logoutUri = client.getLogoutUri();
    String authorizationHeader = null;

    Map<String, String> paramMap = new HashMap<>();
    paramMap.put("tokenId", token.getTokenId());
    paramMap.put("userName", token.getUserName());

    HttpPost post = buildHttpPost(logoutUri, paramMap, authorizationHeader);
    executePostAndParseResult(post, Object.class);
}
```

```
    private HttpPost buildHttpPost(String reqUrl, Map<String, String> paramMap, String authorizationHeader) {
        //
        HttpPost post = new HttpPost(reqUrl);
        if (authorizationHeader != null) {
            //
            post.addHeader("Authorization", authorizationHeader);
        }

        List<NameValuePair> urlParameters = new ArrayList<>();
        for (Map.Entry<String, String> entry : paramMap.entrySet()) {
            urlParameters.add(new BasicNameValuePair(entry.getKey(), entry.getValue()));
        }

        try {
            post.setEntity(new UrlEncodedFormEntity(urlParameters));
        } catch (UnsupportedEncodingException e) {
            log.error(e.getMessage(), e);
        }
        return post;
    }

    private <T> T executePostAndParseResult(HttpPost post, Class<T> clazz) {
        //
        T result = null;
        try {
            //
            HttpClient client = HttpClientBuilder.create().build();

            HttpResponse response = client.execute(post);
            BufferedReader rd = new BufferedReader(
                new InputStreamReader(response.getEntity().getContent()));

            StringBuffer resultBuffer = new StringBuffer();
            String line = "";
            while ((line = rd.readLine()) != null) {
                resultBuffer.append(line);
            }
            log.debug("₩n## response body : '{}'", resultBuffer.toString());

            ObjectMapper mapper = new ObjectMapper();
            result = mapper.readValue(resultBuffer.toString(), clazz);
        } catch (IOException e) {
            log.error(e.getMessage(), e);
        }

        return result;
    }

    private int removeAccessTokens(String userName) {
        //
        return accessTokenRepository.deleteByUserName(userName);
    }
}
```

접근 토큰을 조회하는 getAccessToken() 메소드는 아래 작업들을 처리하도록 구현하였습니다.

1. 인자로 받은 접근 토큰 값을 MD5 해쉬 값으로 변환합니다.
2. 변환된 값과 클라이언트 아이디 인자로 AccessTokenRepository 객체의 findByTokenIdAndClientId() 메소드를 호출하여 데이터베이스에 저장되어 있는

AccessToken 객체를 조회합니다.

3. 조회한 AccessToken 객체를 리턴합니다.

SSO로 로그인한 사용자의 로그아웃을 처리하는 logoutAllClients() 메소드는 아래 작업들을 처리합니다.

1. AccessTokenRepository 객체의 findByUserName() 메소드를 사용해서 사용자 이름으로 발급된 AccessToken 객체들을 조회합니다.
2. 조회한 AccessToken 객체의 클라이언트 아이디로 ClientRepository의 findOne() 메소드를 호출하여 Client 객체를 조회합니다.
3. Client의 logoutUri로 토큰 아이디와 사용자 이름을 포함한 Http Post 요청을 전송합니다.
4. AccessTokenRepository 객체의 deleteByUserName() 메소드를 호출하여 저장된 접근 토큰들을 삭제합니다.

4.1.11. 확장한 Endpoint를 위한 웹 컨트롤러 클래스

사용자 정보 조회, 로그아웃 Endpoint를 위한 웹 컨트롤러 클래스는 아래와 같습니다.

```
@Controller
public class SsoController {
  //
  @Autowired
  private SsoService ssoService;

  @RequestMapping(value="/userInfo", method=RequestMethod.POST)
  @ResponseBody
  public UserInfoResponse userInfo(@RequestParam(name="token") String token,
      @RequestParam(name="clientId") String clientId) {
    //
    AccessToken accessToken = ssoService.getAccessToken(token, clientId);

    UserInfoResponse response = new UserInfoResponse();
    if (accessToken == null) {
      //
      response.setResult(false);
      response.setMessage("사용자 정보를 조회할 수 없습니다.");
    }
    else {
      //
      response.setUserName(accessToken.getUserName());
    }

    return response;
  }

  @RequestMapping(value="/userLogout", method=RequestMethod.GET)
  public String userLogout(@RequestParam(name="clientId") String clientId,
      HttpServletRequest request) {
    //
    String userName = request.getRemoteUser();
    String baseUri = ssoService.logoutAllClients(clientId, userName);

    request.getSession().invalidate();

    return "redirect:" + baseUri;
  }
}
```

접근 토큰을 발급받은 클라이언트의 사용자 정보 조회를 처리하는 userInfo() 메소드는 SsoService의 getAccessToken() 메소드를 호출하여 AccessToken 객체를 조회한 후 그 결과를 UserInfoResponse 객체에 설정하여 리턴합니다.

클라이언트의 로그아웃 요청을 처리하는 userLogout() 메소드는 SsoService의 logoutAllClients() 메소드를 호출한 후 세션의 invalidate() 메소드를 호출하여 사용자 브라우저의 인증된 세션을 무효화합니다.

4.2. 클라이언트 시스템 구축

클라이언트 시스템에서 SSO 환경을 구축하기 위해 구현해야 할 항목들은 아래와 같습니다.

- SSO 요청 시 SSO 서버의 인가(Authorization) Endpoint로 리다이렉트 하기 위한 웹 요청 처리

- SSO 서버의 클라이언트 인가 처리 후 인가 코드 값을 포함하는 리다이렉트 웹 요청 처리
- 사용자의 로그아웃 웹 요청 처리
- SSO 서버의 로그아웃 웹 요청 처리

위에서 기술한 웹 요청 처리를 스프링 MVC로 어떻게 구현했는지 순서대로 살펴보겠습니다.

4.2.1. 클라이언트 인가 Endpoint로 리다이렉트하기 위한 웹 요청 처리

```
@RequestMapping(value="/sso", method=RequestMethod.GET)
public String sso(HttpServletRequest request) {
        //
        String state = UUID.randomUUID().toString();
        request.getSession().setAttribute("oauthState", state);

        StringBuilder builder = new StringBuilder();
        builder.append("redirect:");
        builder.append("http://localhost:8080/oauth/authorize");
        builder.append("?response_type=code");
        builder.append("&client_id=");
        builder.append(getOAuthClientId());
        builder.append("&redirect_uri=");
        builder.append(getOAuthRedirectUri());
        builder.append("&scope=");
        builder.append("read");
        builder.append("&state=");
        builder.append(state);

        return builder.toString();
}
```

SSO 서버로 리다이렉트하기 전 상태(state) 값을 랜덤하게 생성한 후 세션에 "oauthState"를 키 값으로 이를 저장합니다. response_type, client_id, redirect_uri, scope, state 파라미터들을 설정한 리다이렉트 URI를 구성한 후 리턴합니다.

4.2.2. 인가 코드 값을 포함하는 리다이렉트 웹 요청 처리

```
@RequestMapping(value="/oauthCallback", method=RequestMethod.GET)
public String oauthCallback(@RequestParam(name="code") String code,
                @RequestParam(name="state") String state,
                HttpServletRequest request, ModelMap map) {
        //
        String oauthState =
        (String)request.getSession().getAttribute("oauthState");
        request.getSession().removeAttribute("oauthState");

        TokenRequestResult tokenRequestResult = null;
        if (oauthState == null || oauthState.equals(state) == false) {
                //
                tokenRequestResult = new TokenRequestResult();
                tokenRequestResult.setError("not matched state");
        }
        else {
                tokenRequestResult =
                oauthService.requestAccessTokenToAuthServer(code, request);
        }

        if (tokenRequestResult.getError() == null) {
                return "redirect:/me";
        }
        else {
                map.put("result", tokenRequestResult);
                return "authResult";
        }
```

SSO 서버로부터 리다이렉트된 이 요청은 code값과 state값이 파라미터로 넘어옵니다. 먼저, 세션에 저장한 oauthState값과 SSO 서버로부터 넘어온 state값이 동일한지 체크합니다. 동일한 경우에만 code값과 request 값을 인자로 AuthService의 requestAccessTokenToAuthServer() 메소드를 호출합니다.

```
public TokenRequestResult requestAccessTokenToAuthServer(String code,
      HttpServletRequest request) {
  //
  TokenRequestResult tokenRequestResult = requestAccessTokenToAuthServer(code);

  if (tokenRequestResult.getError() != null) {
    return tokenRequestResult;
  }

  UserInfoResponse userInfoResponse =
      requestUserInfoToAuthServer(tokenRequestResult.getAccessToken());
  if (userInfoResponse.getResult() == false) {
    tokenRequestResult.getError(userInfoResponse.getMessage());
    return tokenRequestResult;
  }
  User user = userService.getUser(userInfoResponse.getUserName());
  request.getSession().setAttribute("user", user);

  userService.updateTokenId(user.getUserName(),
      extractTokenId(tokenRequestResult.getAccessToken()));

  return tokenRequestResult;
}
```

AuthService의 requestAccessTokenToAuthServer() 메소드에서는 아래 과정으로 요청을 처리합니다.

1. code값을 인자로 requestAccessTokenToAuthServer() 메소드를 호출하여 SSO 서버의 토큰 Endpoint로 요청을 전송하고 그 결과를 TokenRequestResult 객체로 받아옵니다.
2. 위 요청에 오류가 있는 경우 TokenRequestResult 객체를 바로 반환합니다.
3. SSO 서버의 토큰 Endpoint에서 발급받은 접근 토큰을 인자로 requestUserInfoToAuthServer() 메소드를 호출하여 사용자 정보(User Info) Endpoint로 요청을 전송하고 그 결과를 UserInfoResponse 객체로 받아옵니다.
4. 위 요청에 오류가 있는 경우 TokenRequestResult 객체에 에러 메시지를 설정한 후 이를 바로 반환합니다.
5. UserService의 getUser() 메소드를 호출하여 User 객체를 얻어온 후 세션에 이를 저장합니다.
6. 발급 받은 접근 토큰 값을 MD5 해쉬한 후 UserService의 updateTokenId() 메소드를 호출하여 데이터베이스에 저장합니다.

SSO 서버의 토큰, 사용자 정보 Endpoint로 요청을 전송하고 그 결과를 얻어오는 requestAccessTokenToAuthServer(), requestUserInfoToAuthServer() 메소드의 내용은 아래와 같습니다.

```
private TokenRequestResult requestAccessTokenToAuthServer(String code) {
        //
        String reqUrl = "http://localhost:8080/oauth/token";
        String authorizationHeader = getAuthorizationRequestHeader();

        Map<String, String> paramMap = new HashMap<>();
        paramMap.put("grant_type", "authorization_code");
        paramMap.put("redirect_uri", getOAuthRedirectUri());
        paramMap.put("code", code);

        HttpPost post = buildHttpPost(reqUrl, paramMap, authorizationHeader);

        TokenRequestResult result = executePostAndParseResult(post,
        TokenRequestResult.class);

        return result;
}
```

```
private User requestUserInfoToAuthServer(String token) {
        //
        String reqUrl = "http://localhost:8080/userInfo";
        String authorizationHeader = null;

        Map<String, String> paramMap = new HashMap<>();
        paramMap.put("token", token);
        paramMap.put("clientId", getOAuthClientId());

        HttpPost post = buildHttpPost(reqUrl, paramMap, authorizationHeader);

        User result = executePostAndParseResult(post, User.class);

        return result;
}
```

두 메소드의 정의에서 주목할 부분은 토큰 Endpoint 요청 시 클라이언트 인증을 위한 Authorization 헤더를 설정했다는 것입니다.

4.2.3. 사용자의 로그아웃 웹 요청 처리

사용자로부터 로그아웃 요청이 들어오면 클라이언트 아이디를 파라미터로 설정하여 SSO 서버의 로그아웃(Logout) Endpoint로 리다이렉트 합니다.

```
@RequestMapping(value="/logout", method=RequestMethod.GET)
public String logout(HttpServletRequest request) {
  //
  return "redirect:http://localhost:8080/userLogout?clientId=" +
      getOAuthClientId();
}
```

4.2.4. SSO 서버의 로그아웃 웹 요청 처리

```
@RequestMapping(value="/logout", method=RequestMethod.POST)
@ResponseBody
public Response logoutFromAuthServer(
    @RequestParam(name="tokenId") String tokenId,
    @RequestParam(name="userName") String userName,
    HttpServletRequest request) {
  //
  Response response = oauthService.logout(tokenId, userName);
  return response;
}
```

SSO 서버로부터 로그아웃 요청이 들어오면 AuthService의 logout() 메소드를 호출하여 다음과 같은 작업을 진행합니다.

1) 사용자정보 조회
2) 접근 토큰 정합성 체크
3) 조회된 사용자에 대한 접근 토큰 정보 삭제

```
public Response logout(String tokenId, String userName) {
	//
	Response response = new Response();

	User user = userService.getUser(userName);
	if (user == null || user.getTokenId() == null) {
		//
		return response;
	}

	String savedTokenId = user.getTokenId();
	if (tokenId.equals(savedTokenId) == false) {
		//
		return response;
	}
	userService.updateTokenId(userName, null);

	return response;
}
```

5. 맺음말

지금까지 OAuth 2를 이용한 SSO 구축에 앞서 OAuth 2에 대해 살펴본 후 SSO 환경 구성을 위해 OAuth 2를 어떻게 확장하면 될지 설명한 후 실제로 스프링 부트와 스프링 시큐리티 OAuth로 이를 어떻게 구축하는지 설명하였습니다.

참고 문헌 및 사이트

Ryan Boyd. (2012). **안전한 API인증과 권한 부여를 위한 클라이언트 프로그래밍 OAuth 2.0** (이정림 옮김). 서울: 한빛미디어.

Justin Richer, Antonio Sanso. (2016). **OAuth 2 in Action**. Manning Publications

"The OAuth 2.0 Authorization Framework" .October, 2012. **https://tools.ietf.org/html/rfc6749**

"An Introduction to OAuth 2". July 21, 2014.
https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2

"OAuth 2.0 API". **http://developer.okta.com/docs/api/resources/oauth2.html#basic-flows**

"Understanding OAuth2". January 22, 2016.
http://www.bubblecode.net/en/2016/01/22/understanding-oauth2/