https://brunch.co.kr/@sbcoba/4

**Spring Boot로 만드는 OAuth2 시스템 4**

간단한 OAuth2 서버 만들어 보기

By [이수홍](https://brunch.co.kr/@sbcoba) . Feb 13. 2016

이번  포스팅부터 본격적으로 OAuth2 서버를 만들어 보겠다.

간단한 세팅을 시작으로 하나하나 점증적으로 확장하는 형태로 진행할 예정이다.

(샘플 소스: <https://github.com/sbcoba/spring-boot-oauth2-sample/tree/master>)

**Client ID 계정 생성**

먼저 기본적인 OAuth2 서버에서 가지는 Client  ID와 Client Secret을 아래에서 설정 정보를 추가해보자.

([실제 소스](https://github.com/sbcoba/spring-boot-oauth2-sample/blob/master/src/main/resources/application.properties))

# resources/application.properties  
security.oauth2.client.client-id=foo  
security.oauth2.client.client-secret=bar  
# client-id : client를 식별하는 고유 정보  
# client-secret : 액세스 토큰을  교환하기 위한 비공개 정보 ( 보통 암호 )

**Client의 의미**

OAuth2 서버를 통해 API에 접근을 허가한 클라이언트를 지칭하는 명칭이다.

클라이언트 종류로는 보통 웹, 아이폰 앱, 안드로이드 앱, PC 앱 등이 있다.

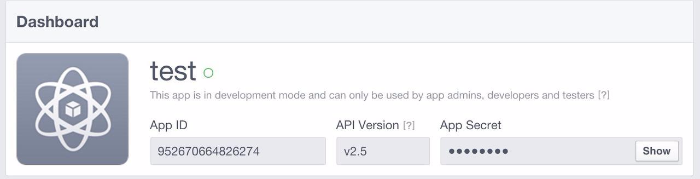
예를 들면 페이스북에서도 [개발자 사이트에서](https://developers.facebook.com/apps) 클라이언트 계정을 아래의 버튼으로 발급할 수 있다.

https://t1.daumcdn.net/thumb/R1280x0/?fname=http://t1.daumcdn.net/brunch/service/user/so6/image/9RyZGAEB9zQE63TDcJSH4I1BxHQ.jpeg

페이스 북에서는 스프링 시큐리티 OAuth2의 명칭과 약간 다르지만 동일한 의미를 지닌다고 생각하면 된다.

Client ID -> App  ID

Client Secret -> App Secret

발급된 페이스북 App 계정

보통 OAuth2 인증을 지원하는 웹사이트에서는 클라이언트 계정을 발급해야 하는 관리자 페이지와 그것을 저장하는 저장소 등을 가지고 있는 경우가 많다. 클라이언트 정보 저장소로는 보통 Mysql과 같은 RDB과  NoSQL 같은 외부 저장소를 이용해서 저장해둔다. 하지만 이 같은 경우에는 가장 간단한 샘플 형태이기 때문에 직접 하드 코딩해두는 경우이다.

그래서 뒷부분에서는 확장하는 부분을 설명하면서 다른 저장소를 이용해서 클라이언트 정보를 관리하는 방법도 다룰 예정이다.

**소스 작성 (**[**실제 소스**](https://github.com/sbcoba/spring-boot-oauth2-sample/blob/master/src/main/java/com/example/DemoApplication.java)**)**

// DemoApplication.java   
@EnableResourceServer // API 서버 인증(또는 권한 설정  
@EnableAuthorizationServer // OAuth2 권한 서버  
@SpringBootApplication  
public class DemoApplication {  
   public static void main(String[] args) {  
      SpringApplication.run(DemoApplication.class, args);  
   }  
}

OAuth2 인증서버와 API 서버를 만들었다. ( 어노테이션 두개로 말이다!! )

놀랍지 않은가? 어노테이션 두개만  설정했을 뿐인데 설정이 완료되었다. 이것이 바로 스프링 부트의 힘이다!

사실 이번 포스팅에서 코딩은 여기 까지다. ( 테스트용 부분은 제외 )

정말 OAuth2 인증이 되는지 한 번 테스트해보자.

**OAuth2의 Access Token 발급받기 테스트**

테스트를 위해 추천하는 툴은 Chrome의 앱인 [Postman](https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop)을 추천한다. 무료인데도 불구하고 API테스트시에는 최고라고 생각된다.   
물론 linux(unix) 계열 운영체제이면 curl로도 충분히 테스트 가능하다. 둘 다 설명하기 힘들기 때문에 텍스트로 설명이 가능한 curl 기준으로 설명할 예정이다. ( [윈도용 curl](http://www.confusedbycode.com/curl/)이 있는데 직접  설치해야 한다.)

스프링 부트에서 스프링 시큐리티 OAuth2의 기본 설정이 클라이언트의 Accees Token 발급 방법을 다섯 가지 방법으로 받을 수 있도록 활성화되어 있다. 그 다섯 가지 방식에 대해 알아 보고 테스트해보려고한다.

일단 위에서 언급한 [github에서 샘플 소스](https://github.com/sbcoba/spring-boot-oauth2-sample/tree/master)를 받고 서버가 실행한 상태에서 테스트가 가능하다.

**1. 권한 코드 방식 (Authorization Code flow)**

보통 서버 사이트 웹에서 인증받을 때 가장 많이 쓰는 방식으로 기본적으로 지원하고 있는 방식이다.

Access Token을 받기 위한 테스트가 다른 방식에 비해 복잡하다.

(따로 인증 관련 요청 페이지 부분을 생성했다.)

먼저 바로 아래의 주소로 브라우저에서 호출한다.

[http://localhost:8080/oauth/authorize?response\_type=code&client\_id=foo&redirect\_uri=http://localhost:8080/test/authorization-code&scope=read](http://localhost:8080/oauth/authorize?response_type=code&client_id=foo&redirect_uri=http://localhost:8080/test/authorization-code&scope=read&state=0807edf7d85e5d)

Client의 접근을 허가 할 것인지 여부를 묻는다.

위 화면에서 Approve체크 후 하단에 Authorize버튼을 클릭하면 아래의 주소로 리다이렉트 되면서 브라우저 화면에서 curl 명령어가 보일 것이다.

http://localhost:8080/test/authorization-code?code=생성된코드

$ curl -F "grant\_type=authorization\_code" -F "code=생성된 코드" -F "client\_id=foo" -F "scope=read" -F "client\_secret=bar" -F "redirect\_uri=http://localhost:8080/test/authorization-code" "http://foo:bar@localhost:8080/oauth/token"

브라우저에 나타나 curl 명령어를 복사해서 실행하면 아래의 Access Token정보가 보일 것이다.

{  
  "access\_token":"1f94c2eb-99bb-412a-bc17-9630b1ae29dc",  
  "token\_type":"bearer",  
  "refresh\_token":"a4b037b7-f736-4bde-a073-7f88279df9bb",  
  "expires\_in":43199,  
  "scope":"read"  
}

**2. 암묵적인 동의 방식 (Implicit Grant flow)**

보통 클라이언트 사이드에서 OAuth2 인증하는 방식이다.

<http://user:test@localhost:8080/oauth/authorize?response_type=token&redirect_uri=http://localhost:8080&client_id=foo&scope=read>

위의 주소 형태를 호출하면 redirect\_uri에 입력된 주소로 리다이렉트 되면서 기본적으로 해쉬태그에서 파라메터로 Access Token을  전달해준다.

**3. 자원 소유자 비밀번호  (Resource Owner Password Credentials flow)**

자원 소유자 즉 사용자의 아이디(username)와 비밀번호로 Access Token 발급한다.

$ curl foo:bar@localhost:8080/oauth/token -d grant\_type=password -d client\_id=foo -d scope=read -d username=user -d password=test

**4. 클라이언트 인증 플로우 (Client Credentials flow)**

클라이언트가 직접  자신의 정보를 통해 Access Token을 발급한다.

$ curl -F "grant\_type=client\_credentials" -F "scope=read" "http://foo:bar@localhost:8080/oauth/token"

**5. Refresh  Token를 통한 Access Token 재발급**

기존에 저장해둔 Refresh Token이  존재할 때 Access Token  재발급받을 필요가 있을 때 사용한다.

그리고 기존 Access Token은 만료된다.

$ curl -F "grant\_type=refresh\_token" -F "scope=read" -F "refresh\_token=발급된 Refresh Token" "http://foo:bar@localhost:8080/oauth/token"

**Access Token을 사용하여 API에 접근 테스트**

위에서 여러 가지 방법으로 발급된 Access Token을 사용해서 API를 호출해보자.

$ curl -H "Authorization: Bearer 발급된 AccessToken" "[http://localhost:8080/members"](http://localhost:8080/members%22)# e.g.  
$ curl -H "Authorization: Bearer 05e63e85-9614-446a-8904-aa6cc556bb1b" "http://localhost:8080/members"

json 정보가  확인되면 성공이다.

다음으로 위 소스에서 설정했던 어노테이션에 대해 한 번  살펴보자.

**@EnableResourceServer**

API 서버를 OAuth2  인증받게 만들도록 하며 하는 역할을 한다. 기본 옵션은 모든 API의 모든 요청에 대해 OAuth2 인증을 받도록 한다.

세부적인 설정을 위해서는 아래와 같이 ResourceServerConfigurerAdapter 클래스를  상속받아서 configure를  구현해야 한다.

// ...  
@EnableResourceServer  
@SpringBootApplication  
public class DemoApplication extends ResourceServerConfigurerAdapter {  
   @Override  
   public void configure(HttpSecurity http) throws Exception {  
      http.authorizeRequests()  
         .antMatchers("/api/\*\*").authenticated();        
   }  
// ...  
}

확장을 하지 않고 기본 옵션은 모든 API는 인증이 필요한 형태로 설정된다.

OAuth2 인증을 확인하기 위하여 OAuth2 토큰 스토어 지정해야 하며, 직접 설정을 하지 않았으면 인메모리 형태로 지정된다. ( 위에서는 지정하지 않았으니 인메모리 형태로 된 상태이다. )  
위와 같이 OAuth2 서버와 API 서버가 같은 곳에서  처리되는 형태라면 같은 기본적으로 인메모리 토큰 스토어를 서로 공유하게 된다.

여기 예제에서는 스프링 부트의 기본 설정을 사용해서 모든 API를 인증받도록 할 예정이다. ( 즉 설정을 안 할 예정이다.)

참고로 스프링 부트에서 기본적으로 설정되는 Class의 위치는 아래와 같다.   
org.springframework.boot.autoconfigure.security.oauth2.resource.OAuth2ResourceServerConfiguration

**@EnableAuthorizationServer**

OAuth2 인증서버를  활성화시켜주는 어노테이션이다.

OAuth2 인증을 위한 AccessToken, RefreshToken 발급과 발급된 토큰을 통한 OAuth2 인증 등 핵심기능을  활성화시켜 준다.

내부에서는 "/oauth/token", "/oauth/authorize" 등 기본적으로 OAuth2에서 사용하는 URI의 접근을 활성화 및 인증 및 내부 예외 처리 기능 등을 가진다.

세부적인 설정을 위해서는 아래와 같이 AuthorizationServerConfigurerAdapter 클래스를  상속받아서 configure를  구현해야 한다.

@EnableAuthorizationServer  
@SpringBootApplication  
public class DemoApplication extends AuthorizationServerConfigurerAdapter {  
// ...  
   @Override  
   public void configure(AuthorizationServerSecurityConfigurer security) throws Exception {  
// OAuth2 인증서버 자체의  보안 정보를 설정하는 부분  
   }  
  
   @Override  
   public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  
// Client 에 대한 정보를  설정하는 부분  
   }  
  
   @Override  
   public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {  
// OAuth2 서버가 작동하기 위한 Endpoint에 대한 정보를 설정  
   }  
   // ...  
}

위와 같이 확장할 수 있지만 여기서는 기본적으로 Spring  Boot에서 기본적으로  설정해주는 형태 그대로 사용할 예정이며 추후 필요한 형태로 확장할 예정이다.

참고로 Spring Boot에서 기본적으로 설정되는 Class의 위치는 아래와 같다.  
org.springframework.boot.autoconfigure.security.oauth2.authserver.OAuth2AuthorizationServerConfiguration

Access Token 발급  테스트하는 부분이  생각보다 많이 길어졌다.

설명은 많이 했지만 사실 코드는 얼마 되지 않는다.

그리고 OAuth2 인증 자체가 복잡한 부분이 있기 때문에 테스트 조차도 복잡해진 부분이 생겼기 때문에 계속 보안할 예정이다.

OAuth2 내부에 구체적인 스펙에 관심이 있으면 국내에 출시된 OAuth2 책이 존재하니까 한 번 읽어보길 권한다. ( 내가 쓴 책은 아니다;; ) <http://www.hanbit.co.kr/ebook/look.html?isbn=9788979149944>

**[안전한 API 인증과 권한 부여를 위한 클라이언트 프로그래밍 OAuth 2.0 - 이북(ebook)](http://www.hanbit.co.kr/ebook/look.html?isbn=9788979149944" \t "_blank)**

[이 책은 OAuth 2.0을 왜 사용해야 하는지, OAuth 1.0을 비롯한 다른 인증 API와 어떤 차이가 있는지, 그리고 어떻게 사용하는지를 알려준다. 단계별 지침과 유형별 예](http://www.hanbit.co.kr/ebook/look.html?isbn=9788979149944" \t "_blank)

[www.hanbit.co.kr](http://www.hanbit.co.kr/ebook/look.html?isbn=9788979149944" \t "_blank)

5편부터는 Spring  Boot에서 지원하는 OAuth기본 설정의 부족한 부분과 확장해야 될 부분 등을  살펴보려고 한다.

<https://brunch.co.kr/@sbcoba/5>

[기술이 세상을 이롭게 변화시킨다2](https://brunch.co.kr/magazine/4programmerss)

# Spring Boot로 만드는 OAuth2 시스템 5

OAuth2 서버를 커스터마이징 해보자 (TokenStore 편)

By [이수홍](https://brunch.co.kr/@sbcoba) . Feb 16. 2016

 앞서 포스팅에서는 최소한의 코드로 OAuth2 서버를 만들어보았다. 사실 설명은 길었으나 정작 코드가 얼마 없는 것을 보고 많이 실망했을 수도 있겠다. 당연히 테스트 형태로 사용할 수는 있어도 실제로 서버로 사용하기에는 많이 부족한 형태이다.

### 실제로 사용하기 부족한 이유를 한번 살펴보자.

1. 기본적으로 설정하지 않으면 **인증**, **권한**, **토큰**, **권한 코드, 클라이언트** 등 영속성이 필요한 정보를 메모리에서 관리한다. 그렇기 때문에 두개 이상의 서버를 같이 운영해서 사용할 수 없는 문제가 생긴다.

(인증과 보안처럼 중요한 서버라면 가용성이 많이 높아야 한다. 최소 두대 이상을 운영할 수 있어야 문제가 발생한 서버가 있더라도 바로 중지되는 사태를 막을 수 있다.)

2. 인증 형태를 모두 Basic 형태로 하기 때문에 실제로 사용하기에는 부족하다.

3. 인증서버와 API 서버가 같이 존재하기 때문에 API 서버를 추가하거나 변경할 때 인증서버도 같이  종료되는 문제가 생긴다.

일단 지금 까지 생각나는 부분은 여기까지이다.

참고로  이야기하자면 스프링 부트를 사용한다고 해서 기존 스프링에 비해 성능이 떨어지거나 내장 톰캣을 사용한다고 해서 기존 톰캣보다 성능이 떨어진다거나 하지는 않는다. 스프링 부트 이름 그대로 해주는 부분은 서버가 시작할 시에 설정 부분을 편하게 하는 역할이 대부분이다. 단순히 그냥 스프링을 사용한다고 생각하면 된다. 그래도 느리다고  생각되면 로직 자체가 느리거나 스프링 자체 문제일 것이다.  
그리고 내장 톰캣은 톰캣 사이트에서 [공식으로 배포하는 톰캣의 내장할 수 있는 버전](http://tomcat.apache.org/download-80.cgi)이다.

(성능상 차이가 없다고 생각하면 된다.)

공식으로 배포하는 내장톰캣

첫 번째로 변경해볼 부분은 토큰 (AccessToken, RefreshToken)을 메모리가 아니라 외부에 저장하는 부분을 진행해보자.

토큰을 저장하는 곳 이름이 **TokenStore**라고 불린다. TokenStore명칭 자체가 토큰을 저장하는 저장소를 의미하기도 하지만 내부에서 사용하는 **인터페이스명을** 지칭하기도 한다.

현재까지 포스팅한 부분까지가 TokenStore는 JVM 내부 메모리를 사용한 저장소를 사용하는 형태이기 때문에 테스트 서버를 재부팅할 때마다 토큰 정보가 초기화된다.   
그래서 발급된 AccessToken(이하  토큰)을 가지고 API 접근 테스트할 때 조차 많이 불편한 상태이다.

이 토큰 정보 부분을 외부 저장소로 저장하는 작업을 해보자. (즉 다른 TokenStore를 사용해보자.)

### 스프링 시큐리티 OAuth2의 기본 TokenStore 종류

1. org.springframework.security.oauth2.provider.token.store.InMemoryTokenStore

- JAVA 내부에서 Map, Queue 구조의 메모리를 사용한 저장소 **(기본)**

2. org.springframework.security.oauth2.provider.token.store.JdbcTokenStore

- JDBC를 사용해서 DB에 저장하는 방식

3. org.springframework.security.oauth2.provider.token.store.JwtTokenStore

- 외부 저장소가 아닌 [JWT](https://jwt.io/)(Token에 JSON 정보를 인코딩 하여 저장하는 방식)를 이용하는 방식

4. org.springframework.security.oauth2.provider.token.store.redis.RedisTokenStore

- [Redis](http://www.redis.io/) ( Key Value 저장소 )에 Token 정보를 저장하는 방식

기본적인 설정은 InMemoryTokenStore를 사용하기 때문에 서버가 리붓되는 동시에 데이터가 초기화된다.

그래서 먼저 TokenStore 중 가장 익숙한 DB를 이용한 JdbcTokenStore를 이용해서 토큰을 저장해 보는 예제를 해보려고 한다.

기존에 H2 DB를 사용하고 있기 때문에 H2 DB를 사용한 예제를 만들어 보려고 한다. 물론 다른 DB를 사용할 때에도 해야 하는 작업에 큰 차이는 나지 않지만 관련 DBMS 설치나 세팅 등 접속하기 전 사전작업을 설명해야 하기 때문에 제외하겠다. (DB를 변경시에도 칼럼명만 바뀌지 않으면 비교적 쉽게 바꿀 수 있다.)

## JdbcTokenStore를 구현하기 위한 사전 준비사항

### 1. OAuth2 토큰을 저장하기 위한 DB 스키마 생성

다행히 스프링 시큐리티 OAuth 프로젝트에서 스키마를 제공하고 있다. ([DB 스키마 링크](https://github.com/spring-projects/spring-security-oauth/blob/master/spring-security-oauth2/src/test/resources/schema.sql))

단 H2 DB의 스키마 형태만 제공하기 때문에 만약 Mysql, 오라클 등 다른 DB를 사용할 때에는 그 DB에 맞도록 적절하게 스키마를 변경해야 할 것이다.( 대신 칼럼명을 변경하면 귀찮아진다. )

위의 스키마 데이터를 복사하여 application.properties 와 같은 경로인 "resources/" 디렉터리 아래에 **schema.sql** 파일을 생성시켜 스키마를 붙여 넣기 한다.   
("**schema.sql"** 파일명과 동일하게 기술해야 따로 설정 없이 관련 서버 시작 시 자동으로 ddl쿼리를 실행한다.)

### 2. 내장 H2 DB를  조회하기 위한 웹 콘솔을 접속하기 위한 세팅

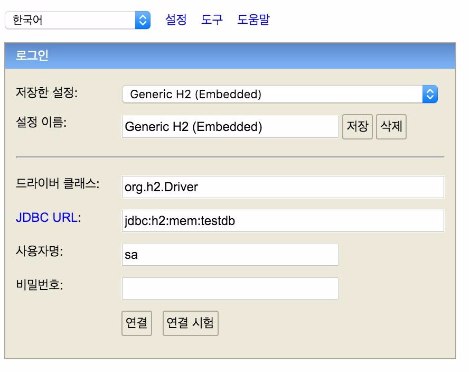
H2 DB에 토큰이  저장되는지 스키마가 생성되었는지 데이터 확인할 수 있는 내장 H2 DB의 웹 콘솔을  활성화시켜보자. (기본적으로는 비활성화)

스프링 부트 1.3 이후부터는 아래와 같이 설정만 해도 바로 H2 console에 접속 가능하다.

# resources/application.properties  
spring.h2.console.enabled=true  
spring.h2.console.path=/h2-console

현재 설명하는 예제는 스프링 부트 1.3 이후 기준이기 때문에 위와 같은 설정만 하면 되지만 그 이전 버전의 스프링 부트에서는 h2 웹 콘솔을 보기 위해서는 [허니몬님 블로그](http://java.ihoney.pe.kr/403)에 설명이 잘되어 있다.

웹 콘솔 활성화 설정이 완료 후 서버를 시작하여 <http://localhost:8080/h2-console> 주소에 접속해본다.

웹 콘솔에 접근하면 위와 같은 화면이 나온다.

위와 같은 화면에 나왔을 때 "JDBC URL" 부분을 **jdbc:h2:mem:testdb**로 **반드시** 바꿔줘야 한다. 이 값이 스프링 부트 안에서 H2 데이터 소스에 접근할 수 있는 (다른 설정을 해주지 않았다면) 기본 접속 정보이다.

스프링 시큐리티를 사용하고 있기 때문에 헤더 부분이 충돌 나서 화면이 보이지 않을 시에는 아래와 같은 설정 정보를 추가해준다.

// <https://github.com/sbcoba/spring-boot-oauth2-sample/blob/master/src/main/java/com/example/DemoApplication.java>  
  
@Override  
public void configure(HttpSecurity http) throws Exception {  
   http.headers().frameOptions().disable();  
   ...  
}

## JdbcTokenStore 설정

앞에서 언급한 설정 부분이 문제가 없이 되었다면 다음과 아래와 같은 설정만  추가해주면 끝이다.

// <https://github.com/sbcoba/spring-boot-oauth2-sample/blob/master/src/main/java/com/example/DemoApplication.java>  
  
@Bean  
public TokenStore JdbcTokenStore(DataSource dataSource) {  
   return new JdbcTokenStore(dataSource);  
}

이 부분만  추가되면 DB에 토큰을 저장할 준비가 완료되었다.   
(너무 쉽게 설정되어 허무할지도 모르지만 이미 지원되고 있는 형태이기 때문에 쉽게 가능하다.)

이제 잘되는지 테스트를 해보자.

## 테스트

테스트 시나리오 아래와 같이 단순하다.

1.  Access Token이  발급해보고 DB를 확인해서 들어갔는지 확인

2. 발급된 Access Token으로 인증이 되는지 확인

Access Token 발급은 이전 포스팅에서 나열한 것 중 가장 단순한 자원 소유자 비밀번호 형태로 하겠다.

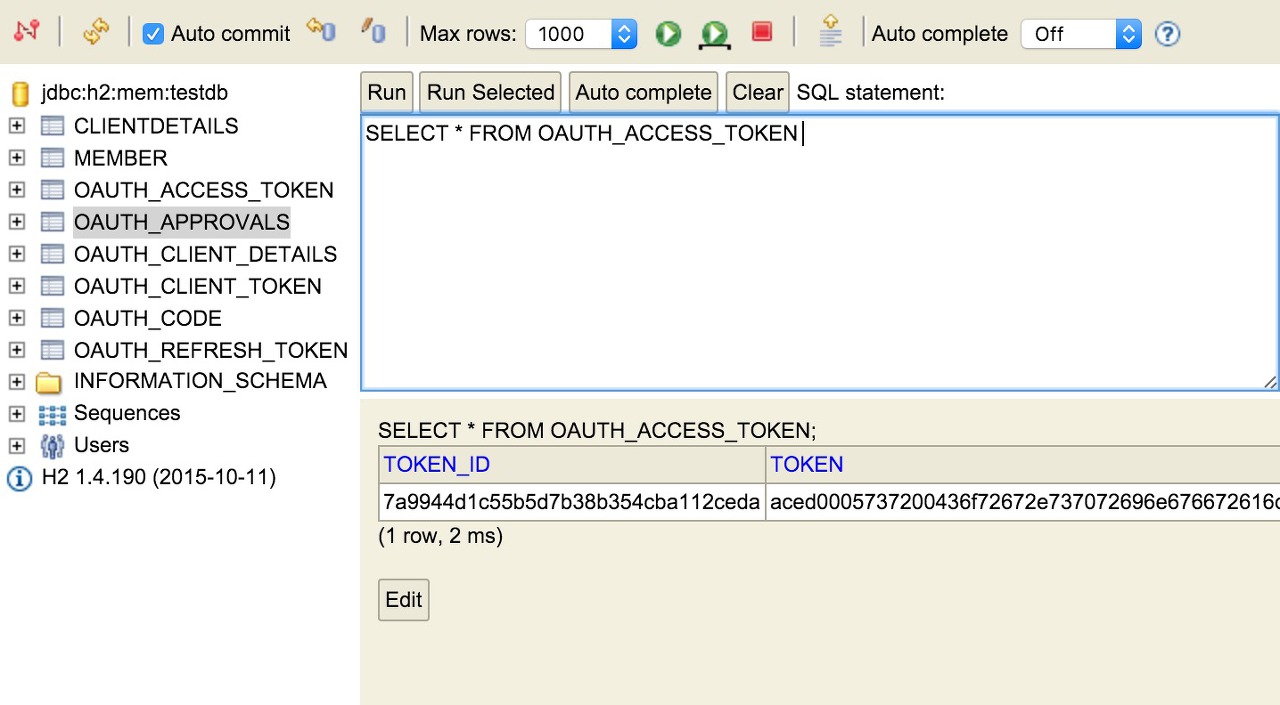
$ curl foo:bar@localhost:8080/oauth/token -d grant\_type=password -d client\_id=foo -d scope=read -d username=user -d password=test

실행시켜 본다.

{  
  "access\_token":"a870ff1f-4c07-4816-b221-73270326ec25",  
  "token\_type":"bearer",  
  "expires\_in":43199,  
  "scope":"read"  
}

자 Access Token을  발급받았다.

DB에 값이 들어갔는지 확인해보겠다. 기본 테이블명은  "OAUTH\_ACCESS\_TOKEN"이다.

OAUTH\_ACCESS\_TOKEN 테이블을 조회 해본다.

위와 같이 쿼리 하면 아래에 토큰이  조회된다.

AccessToken 값 자체에  해당하는 값이 TOKEN\_ID 칼럼이다. 하지만 보이는 값이 다른 이유는 AccessToken 자체도 중요한 값이기 때문에  패스워드처럼 인코딩 해서 보관하기 때문에 그렇다.   
참고로 JdbcTokenStore 클래스의 소스를 살펴보면 MD5 알고리즘을 사용해서 내부에서 인코딩 한다.

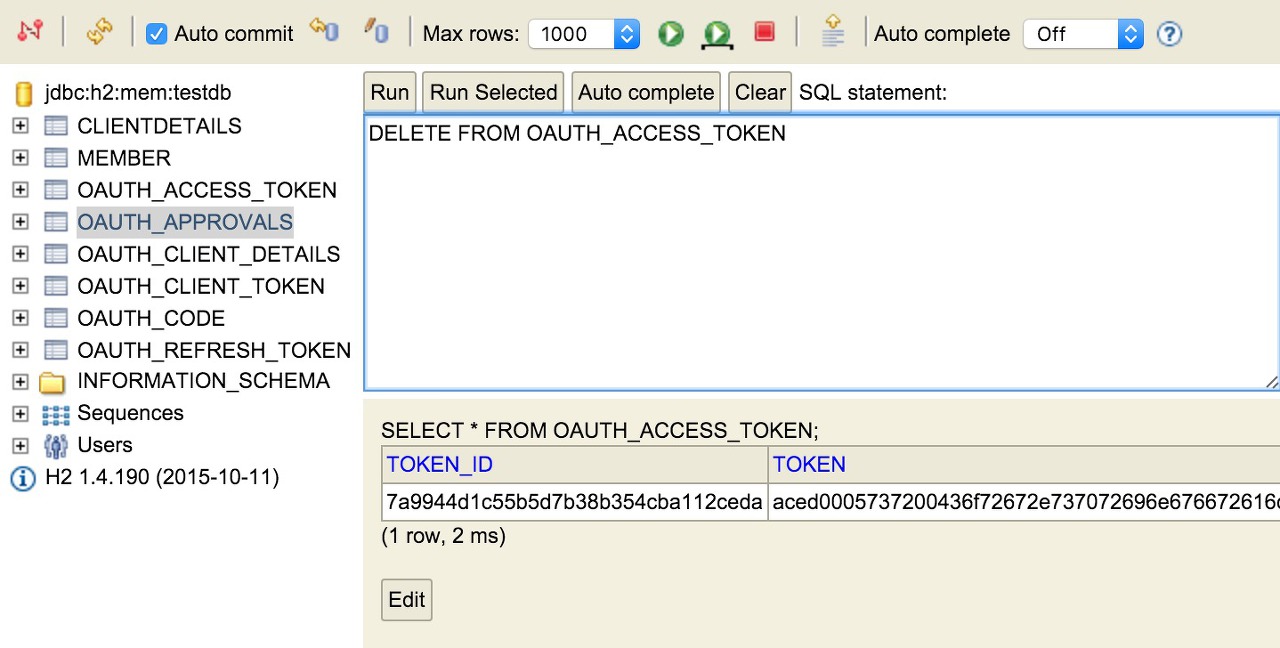
이러한 이유로 AccessToken 값이 일치하지 않는 이유에 대해 특별히 신경을 안 써도 된다.

자 이제 발급받은 AccessToken으로 API를  조회해보자.

$ curl -H "Authorization: Bearer a870ff1f-4c07-4816-b221-73270326ec25" "http://localhost:8080/members"

호출해보니 API가 제대로  조회되는 것을 확인  완료하였다.

그러면 저 DB값을 통해 Access Token이 인증을 받고 있는지 DB에서 데이터를 **제거해보자.**

DELETE 쿼리를 날려보자.

$ curl -H "Authorization: Bearer a870ff1f-4c07-4816-b221-73270326ec25" "http://localhost:8080/members"

앞서 잘되던 요청과 동일한 요청을 다시 한번 날려보자.

{  
  "error":"invalid\_token",  
  "error\_description":"Invalid access token: a870ff1f-4c07-4816-b221-73270326ec25"  
}

예상에 맞게 바로 실패했다.

이로서 비교적 쉽게 DB를 통해 Access Token을 발급하고 사용할 수 있게 되었다.

위의 내용은 미리 알려드린 동일한 [github](https://github.com/sbcoba/spring-boot-oauth2-sample)에 갱신되어 있다.

다음 편을 기대해주시라~

**Spring Boot로 만드는 OAuth2 시스템 6**

API 서버와 OAuth2 서버를 분리

By [이수홍](https://brunch.co.kr/@sbcoba) . Feb 29. 2016

앞서 포스팅 작성한 예제까지는 API 서버와 OAuth2 서버가 하나의 웹 애플리케이션에서 같이 올라가는 형태를 취하고 있다. 예제 자체를 심플하게 유지하려는 목적과 이렇게 개발도 가능하다는 것을 보여주기 위한  것이었다.

하지만 실제 서비스를 하기 위해서는 각각 다른 인스턴스 형태로 서비스를 해야 한다.

이전 포스팅에도 언급했지만 보통 API 서버 같은 경우 자주 갱신되어 배포될 일이 잦기 때문에 OAuth2 인증 서버와 같이 운영하기에는 부담이 있다. OAuth2 인증 서버 같은 경우에는 서비스가 중단되었을 때에는 연관된 모든 서비스가 인증관련으로 문제가 발생하게 된다.

그렇기 때문에 API 서비스의 인스턴스와 OAuth2 서버의 인스턴스를 각각 생성시킬 수 있도록 하여 안정성을 늘려야 한다. 물론 트래픽 분산의 효과도 같이 따라온다.

서문은 길었지만 결론은 제대로 된 서비스 하기 위해서는 API 서버와 OAuth2 서버를 분리해야 된다는 이야기이다.

그리고 아래의 소스는 기존 소스의 **브랜치**로 해두었다. ( [브랜치 링크](https://github.com/sbcoba/spring-boot-oauth2-sample/tree/example6) )

먼저 [기존 소스](https://github.com/sbcoba/spring-boot-oauth2-sample/blob/master/src/main/java/com/example/DemoApplication.java)를 살펴보자.

// <https://github.com/sbcoba/spring-boot-oauth2-sample/blob/master/src/main/java/com/example/DemoApplication.java>  
  
@EnableResourceServer  
@EnableAuthorizationServer  
@SpringBootApplication  
public class DemoApplication extends ResourceServerConfigurerAdapter {  
...  
}

위의 소스에서 어노테이션을 잘  확인해보자.   
**@EnableResourceServer** 어노테이션은 API 서버를 설정하기 위한 부분이다.

**@EnableAuthorizationServer** 어노테이션은 OAuth2 서버를 설정하기 위한 부분이다.

**"extends ResourceServerConfigurerAdapter"** 이름을 보면 **@EnableResourceServer**와 이름이 같은 것을 보면 같은 세트(?)라고 예측할 수 있다.

먼저 위의 설정을 OAuth2를 위한 설정과 API 서버를 위한 설정으로 각각 나누어 보자.

/\*\*  
 \* API 서버  
 \*/  
@Configuration  
@EnableResourceServer  
class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {  
...  
}

/\*\*  
 \* OAuth2 서버 설정  
 \*/  
@Configuration  
@EnableAuthorizationServer  
class AuthorizationServerConfiguration {  
...  
}

한 곳에 있던 설정을 따로 두개의 클래스를 만들어서 도출시켰다.

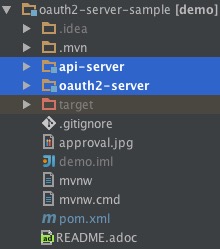
그리고 이전 포스팅에서 설정한 JdbcTokenStore 부분은 OAuth2 서버에서만 사용하게 된다. API 서버에서는 OAuth2 서버의 Token조회 API를 통해서 토큰을 조회하게 된다.

/\*\*  
 \* API 서버  
 \*/  
@Configuration  
@EnableResourceServer  
class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {  
 ...  
}  
/\*\*  
 \* OAuth2 서버 설정  
 \*/  
@Configuration  
@EnableAuthorizationServer  
class AuthorizationServerConfiguration {  
...  
  @Bean  
  public TokenStore JdbcTokenStore(DataSource dataSource) {  
     return new JdbcTokenStore(dataSource);  
  }  
}

그 이외에 설정은 대부분 API 서버 관련 설정이기 때문에 ResourceServerConfiguration 설정 부분으로 옮기면 된다.

그 다음 단계는 이제 프로젝트를 나눌 것이다.

지금 프로젝트에서 oauth2-server, api-server 형태로 폴더를 추가하여 아래와 같은 형태로 만들었다.

폴더 구조

각각  폴더마다 스프링 부트 형태의 구조로 만든 다음 위에서 나눈 설정 형태로 해둔다.

그리고 예제를 로컬에서 실행되기 위해서는 포트가 각각 달라야 한다.   
포트 설정할 때에는 중복만 안되면 된다.   
여기 예제에서는 **OAuth2 서버는 8080 (기본 포트), API 서버는 8081**로 설정 후 사용할 예정이다.   
(만약에 포트가 변경하면 설정 부분에서도 변경이 필요하다.)

OAuth2 서버의 포트: 8080  
API 서버 포트: 8081

**OAuth2 서버 설정 부분**

@EnableAuthorizationServer  
@SpringBootApplication  
public class OAuth2Application {  
   @Bean  
   public TokenStore jdbcTokenStore(DataSource dataSource) {  
      return new JdbcTokenStore(dataSource);  
   }  
// ...  
}

API 서버에서 Token의 정보를 가져가기 위한 요청을  활성화시켜 줘야 한다.   
(기본은  비활성화되어 있다. OAuth2 서버와 API 서버가 같이 있을 때에는 필요 없는 부분이기 때문이다.)

# OAuth2 서버의 application.yml  
...  
# Token 정보를 API(/oauth/check\_token)를 활성화 시킨다. ( 기본은 denyAll )  
security.oauth2.authorization.check-token-access: isAuthenticated()  
...

**API 서버 설정 부분**

@EnableResourceServer  
@SpringBootApplication  
public class ApiApplication {  
  
   @Bean  
   public ResourceServerConfigurerAdapter resourceServerConfigurerAdapter() {        
      //...  
   }  
//...  
}

전체 적인 구조는 메이븐(maven) 멀티 모듈 형태로 이루어진 형태로 자세한 부분을 [github](https://github.com/sbcoba/spring-boot-oauth2-sample/tree/example6)를 참고하면 된다.

API 서버에서는 따로 OAuth2 서버로부터 Access Token 정보를 얻어 와야 되기 때문에 관련 부분을 설정해야 한다.

# API 서버의 application.yml  
  
# 서버 포트 설정  
server.port: 8081  
  
# OAuth2 서버에서 기본적으로 Token정보를 받아오는 URL  
security.resource.token-info-uri: <http://localhost:8080/oauth/check_token>

**DB 구성**

DB는 이전  포스팅처럼 하나의 인스턴스에서는 편의상 하나의 DB에서 API를 위한 테이블과 OAuth2 서버에서 사용하는 데이터(Access Token관리 등)를 위한 위한 테이블을 관리했지만 **OAuth2 서버와 API 서버가  나누어지면서 DB도 나누려고 한다.** ( 만약 하나로  관리하려고 하면 외부의 DB를 사용해야 한다. )

프로젝트 자체가 나누어진 지기 때문에 따로 설정하지 않아도 각각 H2 DB인스턴스 생성되기 때문에 따로  설정하지 않아도 된다.

**서버 실행**

서버 실행하는 방법은 Maven(이하 메이븐)을 사용하며, 멀티 모듈이기 때문에 아래와 같이 하면 된다.

# 부모 프로젝트 폴더에 들어간 후  
# Oauth2 서버 실행 ( 포트 8080 )  
mvn clean -pl oauth2-server spring-boot:run &  
  
# API 서버 실행 ( 포트 8081 )  
mvn clean -pl api-server spring-boot:run &  
  
# 포트 변경이 필요하면 아래와 같은 옵션 추가 후 서버 실행  
mvn clean -pl api-server spring-boot:run -Dserver.port=9999 &

**테스트**

테스트 방법은 이전 포스팅과 동일하게 진행할 것이다.(어차피 같은 OAuth2 서버이다.)  
다른 점이라면 OAuth2 서버와 API를 호출 시에는 서로 다른 서버를 호출하는 정도이다. (로컬이라면 포트만 다르기 때문에 포트만 변경하여 호출하면 된다.)

**Access Token 발급**

$ curl -F "grant\_type=client\_credentials" -F "scope=read" "http://foo:bar@localhost:8080/oauth/token"

이전 포스팅과 OAuth2 서버와 포트가 같기 때문에 요청 와 응답 부분도 동일하다.

{  
  "access\_token":"6dfb79ab-46cc-49ad-9b46-b4da66e9e103",  
  "token\_type":"bearer",  
  "expires\_in":42760,  
  "scope":"read"  
}

Access Token과 함께 API 서버에서 API 호출

$ curl -H "Authorization: Bearer 6dfb79ab-46cc-49ad-9b46-b4da66e9e103" "http://localhost:8081/members"

이전 포스팅에서 API 호출하는 부분과 동일하지만 **포트 부분이 다른 것**을 반드시 확인해야 한다.

이 모든 소스는 기존 소스의 Github의 브랜치로 해두었다. [소스 링크](https://github.com/sbcoba/spring-boot-oauth2-sample/tree/example6)

**부록**

**H2 외부 DB 인스턴스 실행**

이제까지 포스팅은 H2 DB를 프로젝트에 내장하여 사용하는 형태를 가지고 개발을 하였다. 그런데 H2 DB 자체가 외부로 실행시켜서 접속하는 형태가 있다. 그런 방법을 간단하게 설명해보겠다.

**설치 형태**

**1. OSX 계열 ( homebrew 사용 )**

$ brew install h2  
$ h2

**2. Windows 계열**

<http://www.h2database.com/html/main.html>

이 사이트에서 아래의 영역에 있는 **Windows Installer** 링크를 선택하여  다운로드한다.



**3. 그 이외에 운영체제 (리눅스 계열 및 OSX 포함)**

위의  Windows와 동일한 링크에서 **All Platforms** 링크를 선택하여  다운로드한다.

압축을 푼 후 해당 디렉터리로 이동하여 아래와 같이 실행시키면 된다.

$ ./bin/h2.sh

**서버를 통하여 직접 실행**

서블릿을 통해 할 수 있는 방법도 있으나 스프링 부트를 통하면 더욱 쉽게 H2 서버만 실행할 수 있다. [소스](https://github.com/sbcoba/spring-boot-oauth2-sample/blob/example6/h2-server/src/main/java/com/example/H2Application.java)

//...  
@SpringBootApplication  
public class H2Application {  
  @Bean public DbStarter dbStarter() {   
    return new DbStarter();   
  }   
  @Bean   
  public ServletContextInitializer initializer() {   
    return sc -> {   
      sc.setInitParameter("db.user", "sa");   
      sc.setInitParameter("db.password", "");   
      sc.setInitParameter("db.tcpServer", "-tcpAllowOthers");   
    };   
  }   
// ...  
}

**외부의 H2 DB에 접속방법**

H2 DB에 접속하고자 하는 스프링 부트 애플리케이션은 아래와 같이 설정하여 접속이 가능하다. (접속 정보만 맞으면 어디서나 접속 가능하다. 즉 다른 DB와 접근방법이 동일하다.)

# application.xml  
spring:  
  datasource:  
    url: jdbc:h2:tcp://localhost/~/api;DB\_CLOSE\_DELAY=-1;DB\_CLOSE\_ON\_EXIT=FALSE  
    driverClassName: org.h2.Driver  
    username: sa  
    password:

**Spring Boot로 만드는 OAuth2 시스템 7**

JWT 방식으로 바꿔 보자

By [이수홍](https://brunch.co.kr/@sbcoba) . May 12. 2016

 이전 포스팅에서 OAuth2 기본 Access Token을 사용해서 사용해서 교환하여 인증받는 방식을 이야기하였다.

이렇게 하는 방식에는 단점이 존재한다. Access Token만 교환하기 때문에 그 다시 토큰을 가지고 인증 정보를 조회하기 위해 OAuth2 서버로 다시 요청하여 인증된 정보를 얻어오는 오버헤드가 생기게 된다.

참고로 이야기하자면 API 서버에서는 인증과 관련된 정보를 가지고 있지 않기 때문에 호출시마다 Access Token을 사용해서 OAuth2 서버로 요청하여 정보를 가져온다.  
나중에 사용자가 많아지고 트래픽이 늘어나게 되면 API 서버에서 적절하게 캐시를 사용해서 컨트롤해야 하는 부분도 있다. 하지만 요청한 Access Token이 유효(validation)한지 문제가 없는지 계속 확인해야 하기 때문에 OAuth2 서버에도 주기적으로 체크해줘야 한다. 이 부분을 잘 컨트롤해줘야 많은 트래픽에 견딜 수 있는 인증서버 API 서버를 만들 수 있다.

그러한 부분을 어느 정도 해결해주기 위해서 나온 형태가 JWT(Json Web Token 이하 JWT) 방식이다.

JWT를 여기서 설명하자면 많이 길어지기 때문에 잘 설명되어 있는 [**Outsider님의 블로그**](https://blog.outsider.ne.kr/)를 방문하여 한 번 읽어보길 바란다. (<https://blog.outsider.ne.kr/1069>, <https://blog.outsider.ne.kr/1160> )

내용을 읽어보면 알겠지만 JSON 형식의 데이터를 인코딩 하여 토큰으로 이용하는 형태라고 생각하면 될 것 같다.

이전의 OAuth2 서버가 Access Token만 발급한 후 필요한 정보를 Access Token을 통해 정보를 조회하는 형태라면, JWT는 (JSON형태의) 데이터가 직접 붙어 있는 토큰을 OAuth2 서버에서 발급하는 형태이다.

그래서 직접 다시 Access Token을 이용하여 조회할 필요 없이 JWT 토큰에 붙어 있는 정보를 바로 사용하게 된다.

소스를 살펴보기 전에 발급된 토큰을 살펴보자.

먼저 앞서 포스팅했던 기존 OAuth2 서버에서 Access Token을 발급 형태를 살펴보자.

{  
  "access\_token":"6dfb79ab-46cc-49ad-9b46-b4da66e9e103",  
  "token\_type":"bearer",  
  "expires\_in":42760,  
  "scope":"read"  
}

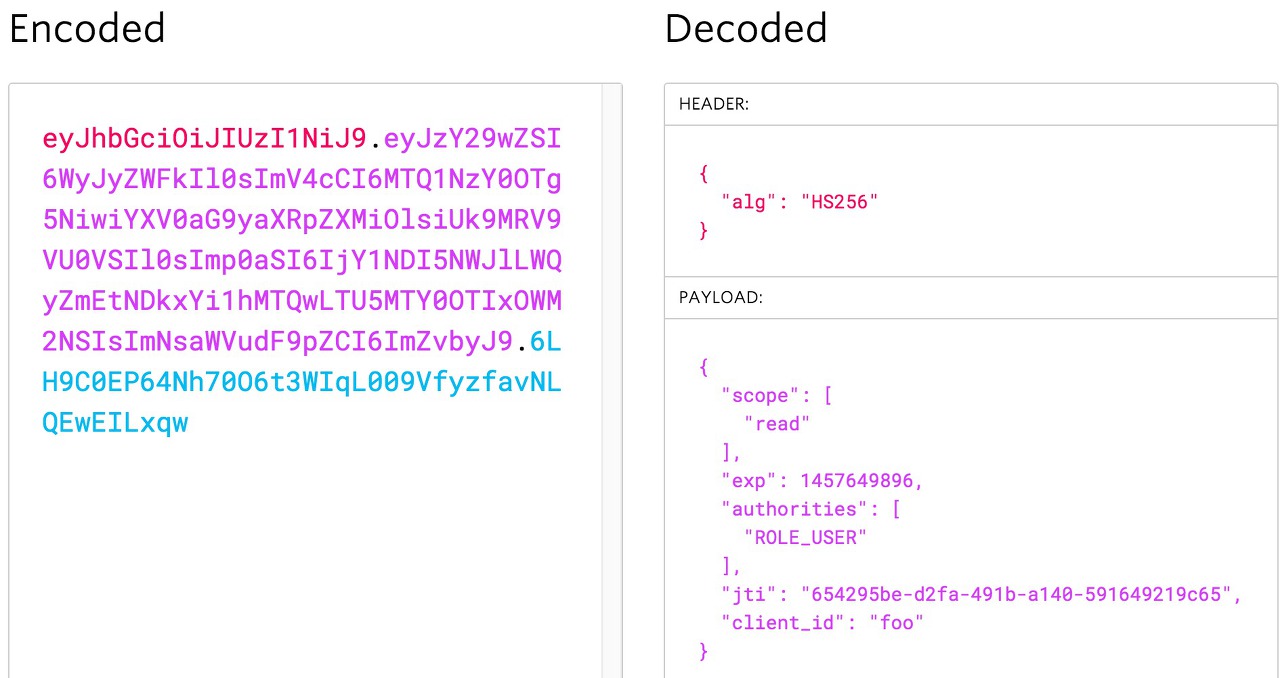
위와 같이 Access Token 정보만 발급하였다.

그럼 JWT토큰을 형태로 OAuth2 서버를 설정했을 때 발급한 형태를 살펴보자.

{  
  "access\_token":"eyJhbGciOiJIUzI1NiJ9.eyJzY29wZSI6WyJyZWFkIl0sImV4cCI6MTQ1NzY0OTg5NiwiYXV0aG9yaXRpZXMiOlsiUk9MRV9VU0VSIl0sImp0aSI6IjY1NDI5NWJlLWQyZmEtNDkxYi1hMTQwLTU5MTY0OTIxOWM2NSIsImNsaWVudF9pZCI6ImZvbyJ9.6LH9C0EP64Nh70O6t3WIqL009VfyzfavNLQEwEILxqw",  
  "token\_type":"bearer",  
  "expires\_in":43199,  
  "scope":"read",  
  "jti":"654295be-d2fa-491b-a140-591649219c65"  
}

위와 구조는 거의 비슷하지만 "access\_token" 안에 이전보다 많은 문자열이 들어가 있는 것이 보일 것이다.

이 문자열이 JWT 데이터이다. 이 데이터를 [jwt.io](http://jwt.io/) 사이트에서 분석해보겠다.



위 문자열에서 JSON 데이터가 도출된다. ( 암호화 형태가 아니라 단순 인코딩 형태이기 때문에 바로 복호화된다. 중요한 정보를 노출하면 안 되는 이유이다. )

이제 JWT 형태로 OAuth2 인증을 받게 하는 형태로 만들어 보자.

먼저 JWT를 사용하기 위해서는 oauth2-server, api-server 의 pom.xml에서 아래와 같은 의존성을 추가해준다.

<!-- oauth2-server/pom.xml, api-server/pom.xml -->  
<dependency>  
   <groupId>org.springframework.security</groupId>  
   <artifactId>spring-security-jwt</artifactId>  
</dependency>

그리고 Access Token을 사용하는 방법이 바뀌었기 때문에 설정 부분도 바꿔 줘야 한다.

먼저 JWT 토큰에서 서명할 때 사용하는 key 값이 필요하다. (JWT는 전송 중에 데이터가 변경되지 않았다는 서명을 가지는데 HMAC이라고 부른다.) 그때 사용할 수 있는 방법이 직접 클라이언트와 서버에 RSA (private, public) 키값을 기술해주거나 서버에서 얻어오는 방법이 있다.   
여기서는 심플하게 서버에서 Key값을 얻어오는 방법으로 하겠다.

# application.yml ( OAuth2 서버 )  
security.oauth2.authorization.token-key-access: isAuthenticated()

위의 설정은 JWT에서 사용된 서명(HMAC) 정보를 검증할 key를 얻어오기 위한 API를 열어주기 위한 설정이다.   
(기본은 설정은 denyAll()로 되어 있다.)  
참고로 기본 Key 값은 OAuth2 서버가 시작할 때 랜덤으로 결정된다.

# application.yml ( API 서버 )  
security.oauth2.jwt.key-uri: <http://localhost:8080/oauth/token_key>

위 설정은 OAuth2 서버에서 열어준 검증용 key값을 얻어오기 위해서 API의 URL을 지정해주는 부분이다. (위 주소(/oauth/token\_key)는 기본값이다.)

이제 소스를 살펴보자.

// OAuth2Application.class  
// ...  
@Configuration  
class JwtOAuth2AuthorizationServerConfiguration extends OAuth2AuthorizationServerConfiguration {  
  
   @Override  
   public void configure(AuthorizationServerEndpointsConfigurer endpoints)  
         throws Exception {  
      super.configure(endpoints);  
      endpoints.accessTokenConverter(jwtAccessTokenConverter());  
   }  
  
   @Bean  
   public TokenStore tokenStore() {  
      return new JwtTokenStore(jwtAccessTokenConverter());  
   }  
  
   @Bean  
   public JwtAccessTokenConverter jwtAccessTokenConverter() {  
      return new JwtAccessTokenConverter();  
   }  
  
}

이전에 있던 JdbcTokenStore 설정 부분을 제거한 후 위 소스 설정을 추가해준다.

간단하게 설명하면 기존 token을 DB로 저장했었던 부분이 JWT로 오면서 없어지게 된다.

위에서 한번 언급했지만 token 자체가 (JSON형태로) 정보를 가지고 있기 때문에 DB에서 읽어 오는 게 아니라 token에 붙어 있는 JSON을 해석해서 토큰 정보를 읽어 오게 된다. 그리고 저장은 Token정보를 JSON으로 바꾼 후 token형태로 발행하게 된다.

그런 부분 설정을 위해서 위와 같은 설정을 하게 되는 것이다.

자 이제 Access Token을 읽어보자.

$ curl foo:bar@localhost:8080/oauth/token -d grant\_type=password -d client\_id=foo -d scope=read -d username=user -d password=test -v  
  
# 결과  
{  
  "access\_token":"eyJhbGciOiJIUzI1NiJ9.eyJleHAiOjE0NjMxMDc1NjcsInVzZXJfbmFtZSI6InVzZXIiLCJhdXRob3JpdGllcyI6WyJST0xFX1VTRVIiXSwianRpIjoiZTAyNmIxNGItMDE4MS00M2U1LTkyOTItYzlhOWI0MDUyZTE4IiwiY2xpZW50X2lkIjoiZm9vIiwic2NvcGUiOlsicmVhZCJdfQ.uxYf\_gC471N14t6HejhS\_Nta9raXdXZ\_zWp9oq4PZfw",  
  "token\_type":"bearer",  
  "refresh\_token":"eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyX25hbWUiOiJ1c2VyIiwic2NvcGUiOlsicmVhZCJdLCJhdGkiOiJlMDI2YjE0Yi0wMTgxLTQzZTUtOTI5Mi1jOWE5YjQwNTJlMTgiLCJleHAiOjE0NjU2NTYzNjcsImF1dGhvcml0aWVzIjpbIlJPTEVfVVNFUiJdLCJqdGkiOiJiMzdiZDE1Ny00NDRmLTQ5ZjEtOTljYy1jYWVkYWNjZTAzZTQiLCJjbGllbnRfaWQiOiJmb28ifQ.ouV83CljkzdRMW7GBQ3EpShUwYocL2cqheF5Pb1ntP0",  
  "expires\_in":43199,  
  "scope":"read",  
  "jti":"e026b14b-0181-43e5-9292-c9a9b4052e18"  
}

위 토큰을 통해서 API를 호출해보자

$ curl http://localhost:8081/members -H "Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJleHAiOjE0NjMxMDc1NjcsInVzZXJfbmFtZSI6InVzZXIiLCJhdXRob3JpdGllcyI6WyJST0xFX1VTRVIiXSwianRpIjoiZTAyNmIxNGItMDE4MS00M2U1LTkyOTItYzlhOWI0MDUyZTE4IiwiY2xpZW50X2lkIjoiZm9vIiwic2NvcGUiOlsicmVhZCJdfQ.uxYf\_gC471N14t6HejhS\_Nta9raXdXZ\_zWp9oq4PZfw"  
  
# 결과  
// API 결과 JSON  ...

API를 호출한 후 OAuth2 서버가 반응하는지 확인해보자. 내 예상이 맞다면 아마 로그가 안 올라올 것이다.

즉 토큰만으로 API를 호출할 수 있게 된 것이다.   
( API 서버에서 Access Token을 확인하기 위한 OAuth2 서버로의 요청이 사라 졌다. token자체에서 데이타를 읽기 때문에! )

전체 소스는 [이 곳](https://github.com/sbcoba/spring-boot-oauth2-sample/tree/example7)에 있다.

## [스프링부트와 OAuth2](http://springboot.tistory.com/34)

<http://springboot.tistory.com/34>

<http://springboot.tistory.com/35>

<http://springboot.tistory.com/36>

<http://springboot.tistory.com/37>