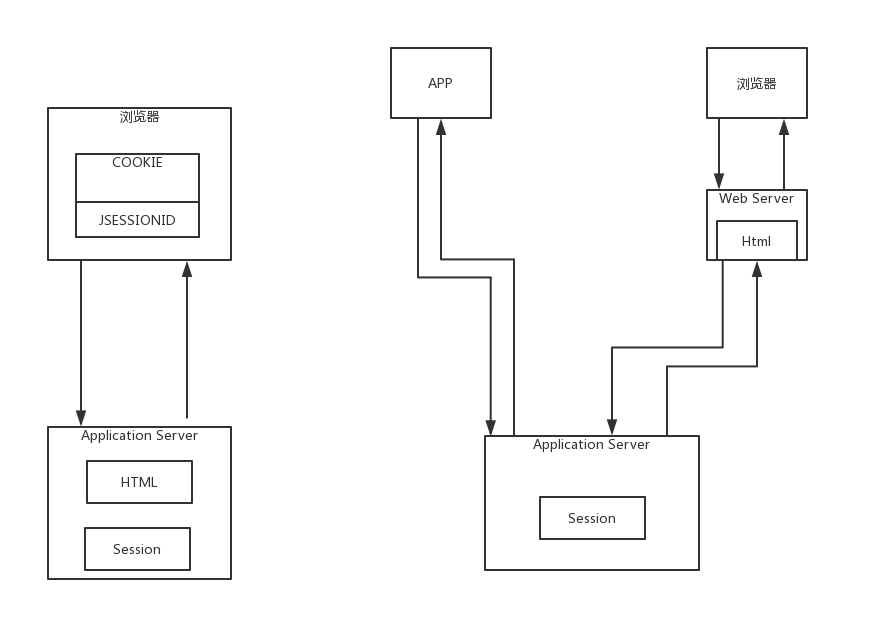
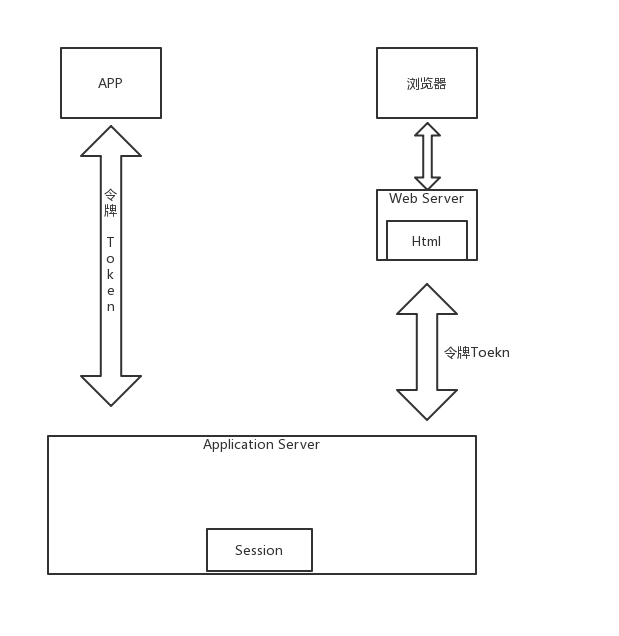
## 5.使用Spring Security Oauth开发APP认证框架

### 5-1.Spirng Security OAuth简介

在之前的登录方式中，当用户登陆之后的信息，都是存在session中的，也就是说，所有的检测都是依赖于session。在这之前章节的服务结构如下图中左边的结构，有用户浏览器发起请求，同时携带cookie到应用服务，应用服务受到请求之后，读取session中的JSESSIONID，在服务器内存中找寻对应的用户信息，之后，如果找到，则直接将用户信息放入security context，如果没找到，则创建一个新的session。基于这样的方式，此时，由于在浏览器中存在有cookie标准，这种实现方式较为简单。



而随着技术的发展，用户的终端选择越来越多，比如app，小程序，等各种各样的前端技术，按照目前比较流行的前后端完全分离的架构，用户直接访问的是web server，包括浏览器的渲染以及ajax请求，都是有web服务器完成的，web server收到请求之后，再发送给application server，不管是app还是web server，当技术演变成这个架构之后，整个流程就出现了根本性的不同，用户不在通过浏览器访问应用服务，而是通过一些第三方的应用，比如app或者web server，此时，再使用cookie session的方式登录，也并不是不行，这时候，需要前端支持模拟cookie标准，但是就算是按照cookie标准，也会带来开发过程繁琐，用户体验不好等影响，甚至于像微信小程序，根本就不提供cookie机制，那么session机制也就无从谈起了。很明显，在这种情况下，在使用session机制未免太过繁琐，此时，我们可以使用token机制来替代整个流程。



如上图所示，使用token来取代cookie的作用，同时，因为token是有应用服务器生成的，所以生成，存储，校验的策略都由应用服务器来完成。可以使用非常多的技术手段来增强token的安全性，例如token刷新机制等等，此时，应用服务器可以很方便的控制token，实际上，这个token本质上就是一个String字符串，使用者在访问应用服务器的时候只需要在发送http请求的时候将Token带上即可，在参数中或者请求头中携带都可以。

至此，应用服务器要完成的角色和功能就很清楚了，应用服务器需要提供接口给使用者获取token，以及提供接口给应用者访问rest服务，讲到这里很自然的发现这种处理方式无非就是OAuth2协议，只不过，从之前小节中，我们开发者写的角色，从第三方应用变成服务提供商。也就是说，这时候，需要登录的终端，按照Oauth2协议，像应用服务器申请授权码，token，以及用户资源，而此时，作为服务商，即应用服务器开发者，我们需要提供两个接口，即引导用户完成授权请求接口返回授权码，获取用户token请求，返回token，获取用户资源请求等，当然，依据终端的不同，可能不一定是跳转页面之后引导用户授权，也可能是直接让用户输入账号密码或者短信认证等，用以获取授权。

此时，整个应用结构就变成了如下图所示：



而在spring security oauth中，则封装好了服务提供商需要完成的大部分业务，使用spring oauth，能够非常快速的开发服务提供商业务，并像第三方应用发送以及认证token，最终返回用户信息。在spring oauth中，并没有明确的规定token如何生成，但是也提供了默认的实现，前期先使用默认的实现，后期也可以使用其他实现方式。

而spring oauth中的资源也是通过spring security来进行管理的，也就是说，会有个oauth2AuthenticationProcessingFilter过滤器，负责处理该功能，从请求中，拿出请求发出的token，根据配置的存储策略，从存储中找到token中对应的用户信息，然后判断返回的用户信息，是否有权限，是否能访问到资源，这样就实现了资源服务器的功能。

在本章中，会解决这些问题

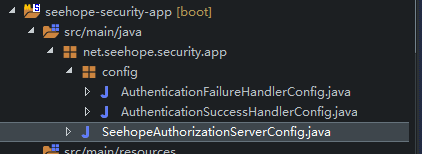
1. 实现一个标准的Oauth2协议中provider角色的主要功能
2. 重构之前三种认证方式的代码，使其支持token
3. 以及spring oauth的高级特型，比如使用jwt生成token，以及基于Oauth协议的单点登录系统等

### 5-2.实现标准的OAuth服务提供商

按照上小结的内容，明确了，如果要实现一个服务提供商的角色，那么实际上就是完成两个服务，即认证服务以及资源服务。

因为这章主要是针对app，或者是前后完全分离的架构，将所有的终端都视作第三方应用，所以，大部分的代码都写在app项目中，先将之前的代码重构一下。找到demo项目中的pom配置，将demo的依赖从browser改成app，即让之前配置的针对于浏览器的配置先失效，同时，将demo项目启动，查看是否能正常启动。在重构的过程中，充分的思考一下那些未引入的bean是应用于全局还是单独应用于brower，在上小结最后提供的代码中，已经将所有的代码都整理好了。

之后，在app项目中新建



至于前两个handlerconfig这里暂时先不用管它，是直接从brower中复制过来的在启动工程的时候，需要这两个bean而已，先来配置认证服务器。

/\*\*

\* <b>Description:写上注解之后即注册了spring oauth 默认定义的4中访问模式

\* 启动之后，会自动往spring容器中添加映射

\* 默认情况下

\* 访问这些URL需要携带有固定参数，具体可参考OAuth协议标准，百度Oauth协议，第4章第1小节中有提到

\* </b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-app

\* <br><b>PackageName:</b> net.seehope.security.app

\* <br><b>ClassName:</b> SeehopeAuthorizationServerConfig

\* <br><b>Date:</b> 2019年3月10日 下午9:55:36

\*/

@Configuration

@EnableAuthorizationServer

public class SeehopeAuthorizationServerConfig {

}

可以看到该类的写法非常简单，就两个注解而已，这时候，在启动服务，发现在日志中出现

2019-03-12 14:49:36.758 INFO 13404 --- [ main] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} inited

2019-03-12 14:49:39.432 INFO 13404 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/authorize],methods=[POST],params=[user\_oauth\_approval]}" onto public org.springframework.web.servlet.View org.springframework.security.oauth2.provider.endpoint.AuthorizationEndpoint.approveOrDeny(java.util.Map<java.lang.String, java.lang.String>,java.util.Map<java.lang.String, ?>,org.springframework.web.bind.support.SessionStatus,java.security.Principal)

2019-03-12 14:49:39.437 INFO 13404 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/authorize]}" onto public org.springframework.web.servlet.ModelAndView org.springframework.security.oauth2.provider.endpoint.AuthorizationEndpoint.authorize(java.util.Map<java.lang.String, java.lang.Object>,java.util.Map<java.lang.String, java.lang.String>,org.springframework.web.bind.support.SessionStatus,java.security.Principal)

2019-03-12 14:49:39.442 INFO 13404 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/token],methods=[GET]}" onto public org.springframework.http.ResponseEntity<org.springframework.security.oauth2.common.OAuth2AccessToken> org.springframework.security.oauth2.provider.endpoint.TokenEndpoint.getAccessToken(java.security.Principal,java.util.Map<java.lang.String, java.lang.String>) throws org.springframework.web.HttpRequestMethodNotSupportedException

2019-03-12 14:49:39.444 INFO 13404 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/token],methods=[POST]}" onto public org.springframework.http.ResponseEntity<org.springframework.security.oauth2.common.OAuth2AccessToken> org.springframework.security.oauth2.provider.endpoint.TokenEndpoint.postAccessToken(java.security.Principal,java.util.Map<java.lang.String, java.lang.String>) throws org.springframework.web.HttpRequestMethodNotSupportedException

2019-03-12 14:49:39.447 INFO 13404 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/check\_token]}" onto public java.util.Map<java.lang.String, ?> org.springframework.security.oauth2.provider.endpoint.CheckTokenEndpoint.checkToken(java.lang.String)

2019-03-12 14:49:39.448 INFO 13404 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/confirm\_access]}" onto public org.springframework.web.servlet.ModelAndView org.springframework.security.oauth2.provider.endpoint.WhitelabelApprovalEndpoint.getAccessConfirmation(java.util.Map<java.lang.String, java.lang.Object>,javax.servlet.http.HttpServletRequest) throws java.lang.Exception

2019-03-12 14:49:39.450 INFO 13404 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/error]}" onto public org.springframework.web.servlet.ModelAndView org.springframework.security.oauth2.provider.endpoint.WhitelabelErrorEndpoint.handleError(javax.servlet.http.HttpServletRequest)

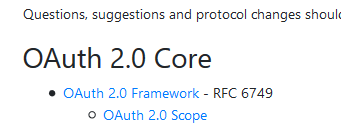
这样的日志，观察红色的字，发现此时spring oauth已经往容器中注入了认证服务接口，此时，整个demo项目已经是一个认证服务器了。

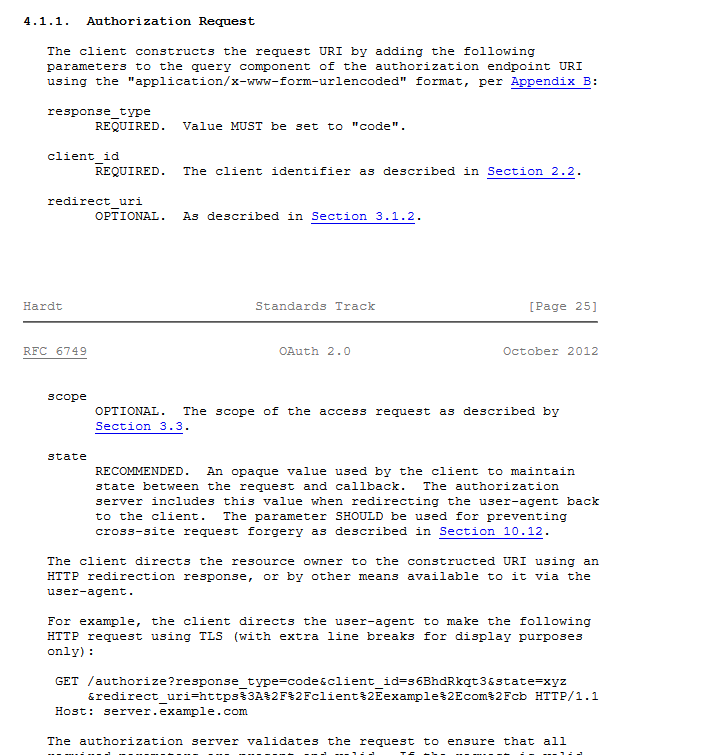
接着，来研究一下如何访问这4中授权模式。

首先，是授权码摸模式，该模式，需要给用户提供两个服务，其一，引导用户完成授权，其二，提供给第三方应用使用授权码交换token地址。

请求中请求"{[/oauth/authorize]}"请求就是提供给第三方应用获取授权码请求，而"{[/oauth/token],methods=[GET]}"请求则是提供给第三方刚应用使用授权码换取AccessToken。

在服务器启动之后，访问第一个请求，在访问的过程中，需要提供一些参数，具体参数和说明可以在百度Oauth2协议中获取。

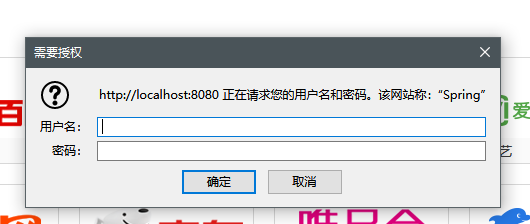




需要提供response\_type 且值必须为code，client\_id 第三方应用id是有由服务器分发的，redirect\_uri，当服务器获取到授权码之后，返回给第三方应用的回调地址，sopce为服务提供商自定义的获取用户信息的权限，此参数也必须要有，最后一个state是推荐携带，在本例中先不携带。同时，在默认配置情况下，程序启动会自动抛出client\_id，可以在控制台打印中找寻该参数，之后，将url拼装好。

localhost:8080/oauth/authorize?response\_type=code&client\_id=e9352d03-36bb-4200-bbdf-ab31e3686af4&redirect\_uri=http://example.com&scope=all

之后，浏览器弹出认证窗口



很明显，这是一个spring security的basic认证，此时，我们的代码将扮演一个服务商的角色，也就是，qq或者微信，对于服务器来说，那么就需要知道几件事，此时，是哪个第三方应用在请求授权，其二，是在请求哪个用户给授权，最后，请求什么样的权限，第一个业务通过client\_id来进行区分，第二，通过用户名密码确认是哪个用户在授权，第三，通过请求中携带的scope参数来明确授予什么权限。Scope是服务提供商自己定义的字符串。

此时，需要用户输入用户名以及密码才能登陆，当用户输入用户名与密码之后，spring security会在调用spring security中注册的userDetailsService的实现，从数据库中查询对应的用户，因为在前面的章节中已经设置好了，用户名为任意，密码固定为123456，且用户权限为admin的固定返回，在这里，如果需要用户能够登陆，这权限列表中必须要有ROLE\_USER，先将该角色加上

return new SocialUser(userId, password, true, true, true, true,

AuthorityUtils.commaSeparatedStringToAuthorityList("admin,ROLE\_USER"));

之后，才能允许用户进行授权登陆，此时，加上权限配置，重新启动服务，当然，此时还会出现一些问题，每次服务器重启都将刷新client\_id，在重启之前，可以先将client\_id以及client\_secret配置在配置文件内。

在配置文件内添加

security:

basic:

enabled: true

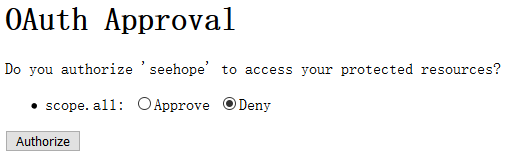
oauth2:

client:

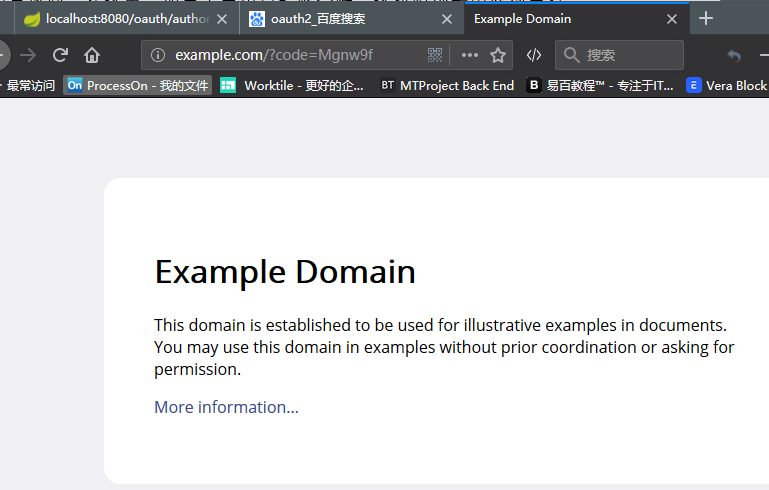
client-id: seehope

client-secret: seehopesecret

之后，通过固定的client-id以及任意用户名，固定密码123456登录，跳转到授权页面。



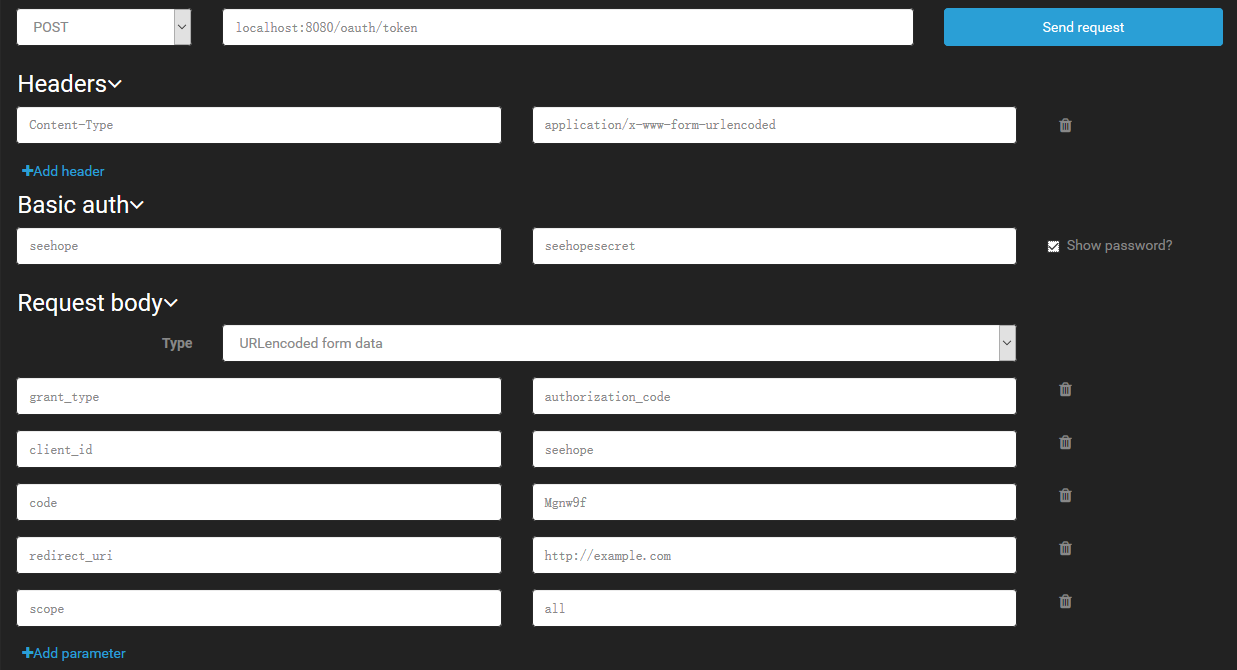
选择授权之后，跳转到填写的redirect\_uri



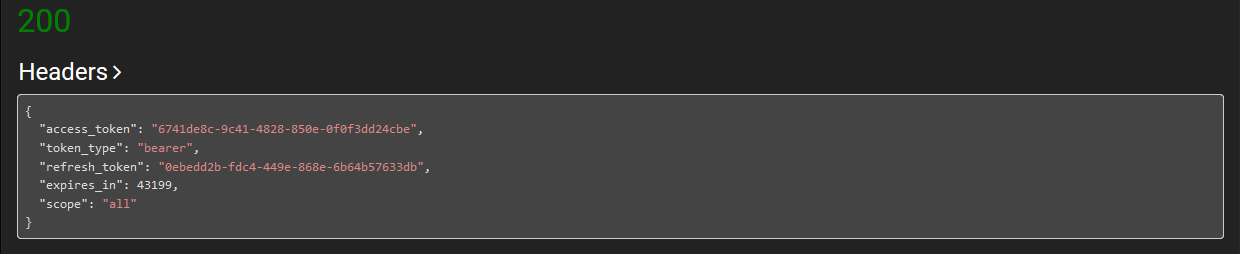
并且携带授权码。Mgnw9f。之后，授权链接测试完成，之后，测试获取token请求地址。{[/oauth/token],methods=[POST]} 因为该地址的访问方式只支持post，所以使用工具，对该url发送请求。当然，访问该url也需要携带一系列的参数，同样在文档中可以查看到参数信息，在4.1.3小节中有说明。

有grant\_type:这里必填authorization\_code，code:在上一步中获取的授权码，client\_id:第三方应用id，redirect\_url:获取到授权之后跳转的地址，socpe:给用户授权的域。

同时，在请求头中，必须要携带有应用认证信息，即client\_id 以及clent\_secret

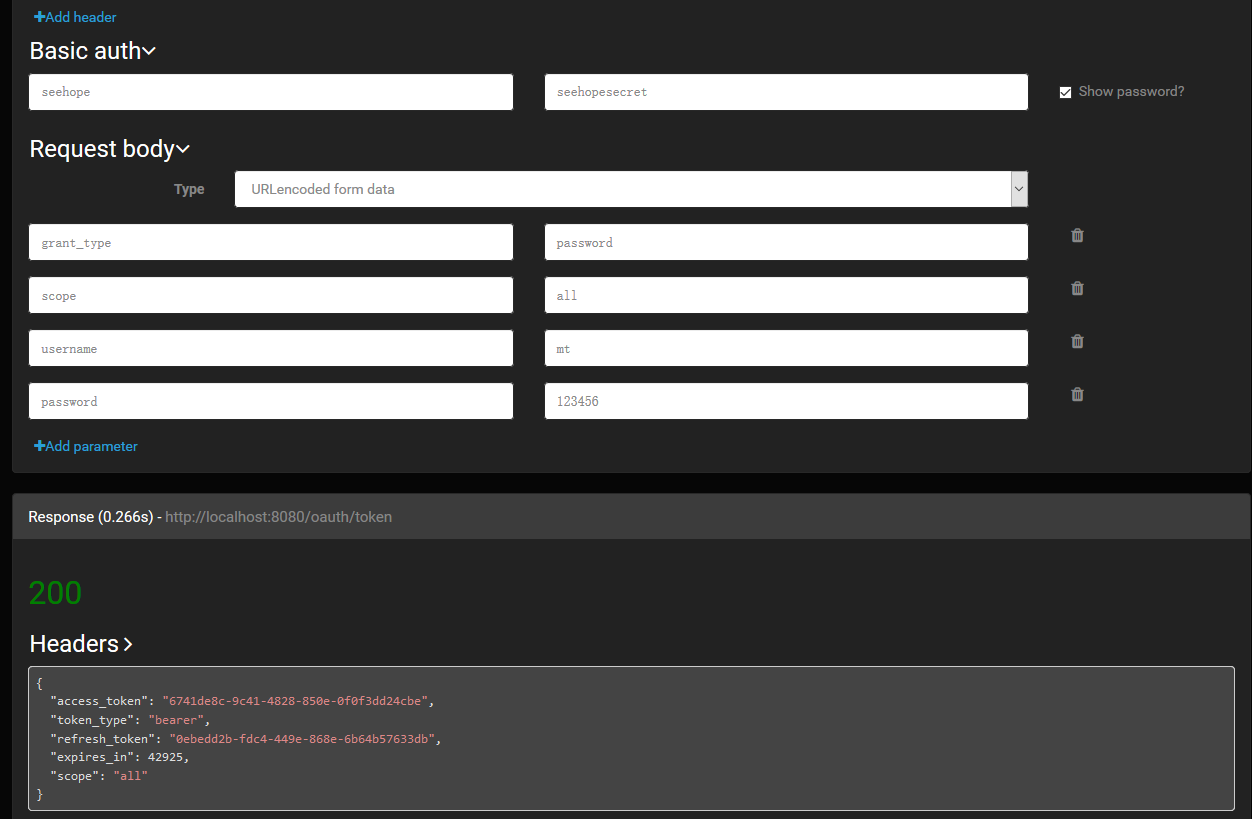


发送之后得到返回结果



之后，整个授权码的授权模式就请求完成了

除此之外，密码授权模式获取token信息的请求地址和本例相同，只需要将其中参数修改一下。



之后，密码授权模式授权完成。成功获取到access\_token，在密码模式下，只需要将用户名密码，发送到认证请求中，就可以获取token授权，但是对于服务提供商而言，无法确认用户名与密码是否真的是用户交给第三方应用的，或者是第三方应用从何渠道而来，不得而知，只要第三方应用有用户名密码，就可以获取授权，这种模式存在一些安全隐患。但是在当前业务场景中，即app登录认证场景，是可以使用密码模式认证的，即用户将用户名密码交给app前端，app前端发送请求到该url给服务器交换token。所以该授权模式可以用在，本公司app以及本公司业务中交换token。同时，如果是同一个用户请求token，那么，无论是授权码模式还是用户名密码模式，获取到的token都是同一组记录，这点可以在截图中看出。只要accessToken未过期。

剩余的两种授权模式在这里就不一一说明了，实际上，整个认证流程四种模式已经实现，包括Token的存储以及校验。整个代码就一个注解@enableAuthorizationServer当然，这里运用的都是spring oauth 2 标准的实现，之后的代码中，还会对该流程进行重构，认证服务器提供完成。

之后，来实现资源服务器。同样的，依然通过注解实现。

@Configuration

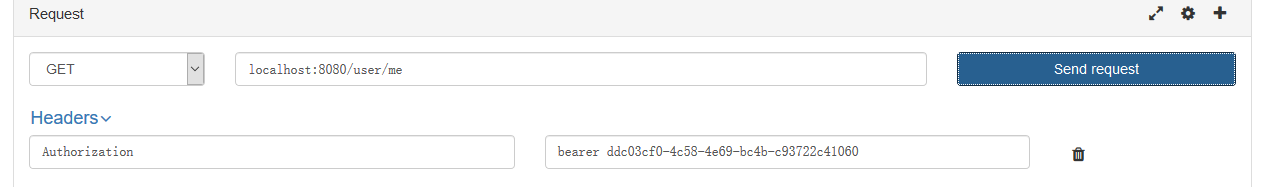
@EnableResourceServer

public class SeehopeResourceServerConfig {

}

此时，demo项目同样实现认证服务器和资源服务器的功能，在加上注解之后，在demo项目中的controller中已经存在有/user/me请求以获取用户资源，在重启服务之后，可以通过该请求获取用户信息。当然，在重启服务期之后，上小结中生成的token已经失效了，需要重新获取token，之后在获取user信息。

在获取用户的信息的时候，需要将token 以及 token type放入请求头中。

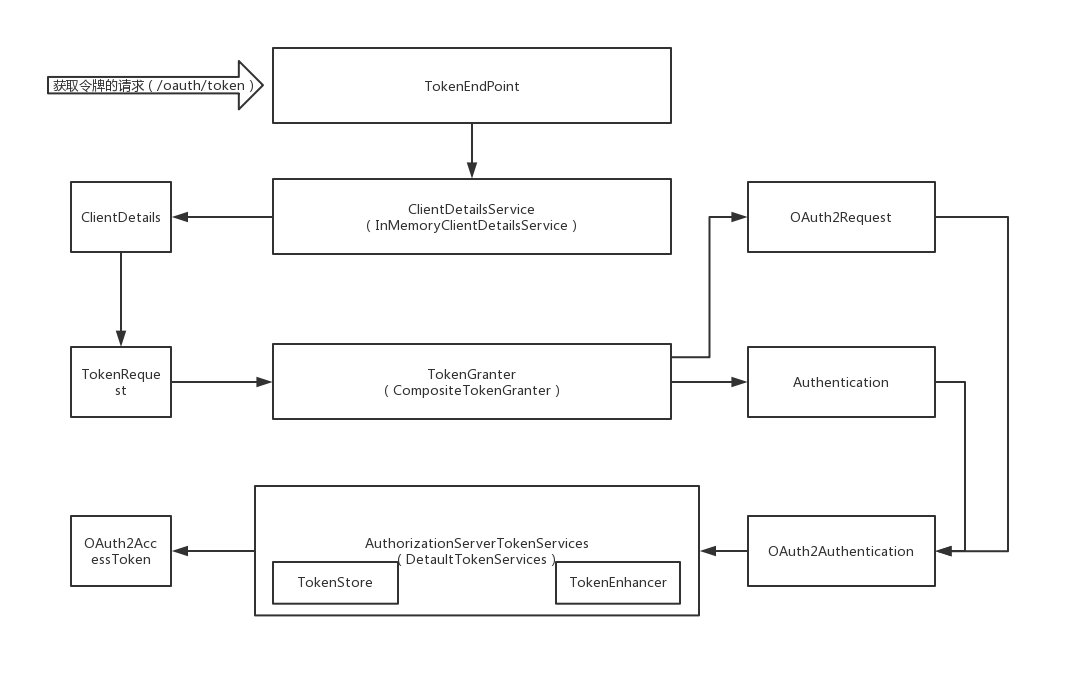




之后，得到返回的用户信息。到此，整个oauth流程就已经完成了。当然，这里使用的都是springsecurity的标准实现实现的，在整个流程中，还存在有很多问题，首先，认证方式只能使用oauth协议提供的标准认证方式，如果想让用户使用手机验证码登录等，在默认的实现中，是应用不了的，同时accessToken目前是存在内存中，当服务器重启之后，所有token失效，此外，token的生成和校验规则也是写死的，能不能自定义生成和校验流程，比如说比较流行的jwt来做token。在之后的小结中，会重新定制针对app的认证授权流程。

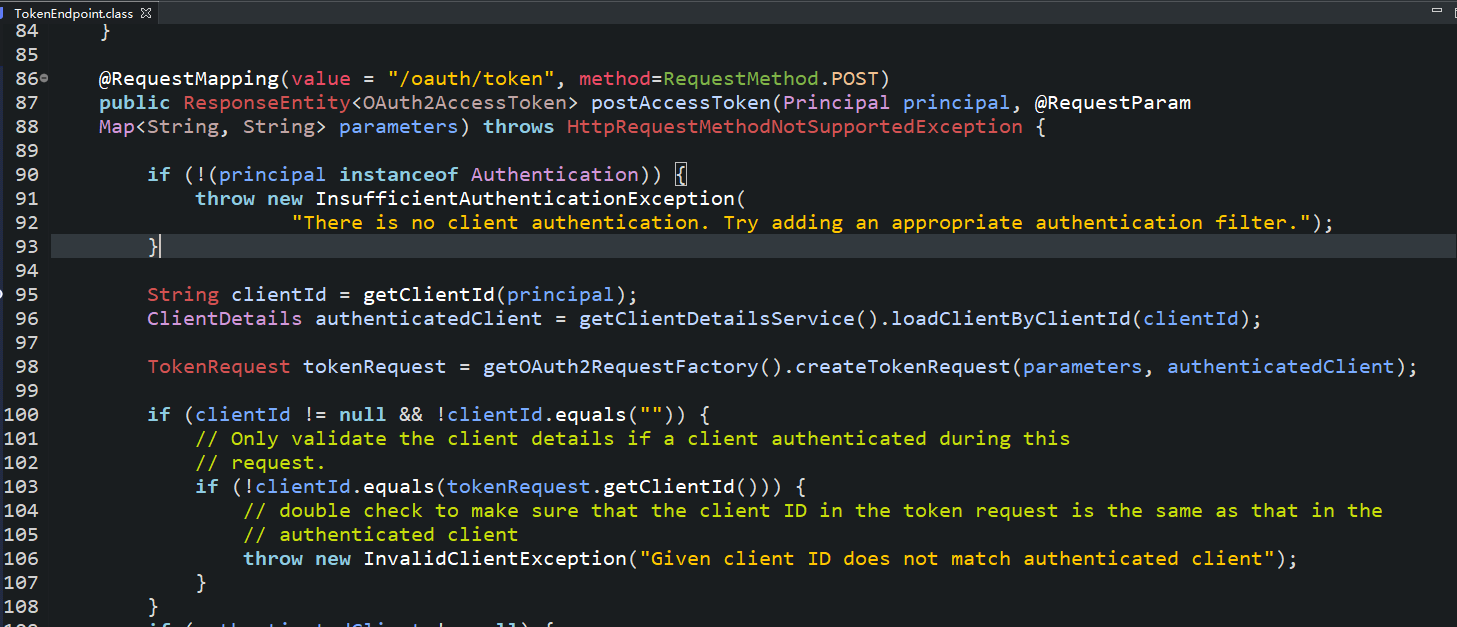
### 5-3.Spirng Security OAuth 核心源码解析

在上小结中，已经实现了oauth服务商的角色，认证服务器和资源服务器，实际上代码非常简单，就是两行注解。实现了标准流程之后，我们需要将自定的认证业务逻辑装配入标准的认证流程，为了实现这样的目的，需要先研究一下spring security oauth源码。

在解释代码之前先来看整个流程架构图，在讲解之前对整个流程有个大概的认知。  


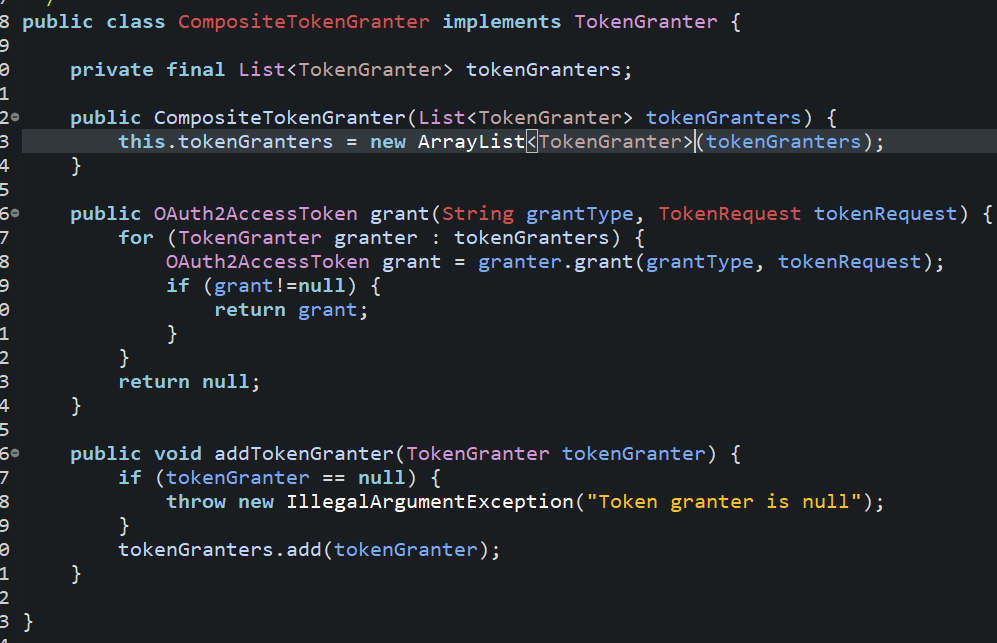
首先，当请求/oauth/token 传入到系统中，将会被TokenEndPoint所拦截，可以将这个类理解成一个controller，该类会调用clientDetailsService服务，因为在访问该请求的时候，无论使用哪种授权模式，都需要传入authorization信息，即appid和appsecret，这两个信息就像用户的用户名以及密码，在服务提供商有记录，同时盖service作用和userDetailsService作用一样，都是从存储中返回信息，只不过这里是返回在服务商中注册了的第三方应用的信息，之后，组装成ClientDetails，之后，将其他访问参数合并，并封装成TokenRequest，即scope，grant\_type，username，password，code等参数，都会封装在该类中，传入给tokenGranter，该类先判断传入的grant\_type取值，之后，根据不同的值，调用不同的授权码验证逻辑，即，验证授权码或者判断用户名密码，如果是用户名密码认证方式，那么这里会调用userDetailsService，也会将返回值拼装为authentication对象，同时，之前的token和clientDetail会被合并为oauth2Request，同时oauth2Request和Authentication又会被合并为Oauth2Authentication,如果所有的验证都通过了，最后会调用AuthorizationServerTokenServices,该类负责对token进行生成以及存储验证等业务，同时，这里也开放了两个接口TokenStore以及TokenEnhancer，可以覆盖token的存储以及生成方式，最终，才会生成OAuth2AccessToken。

最终将服务启动，在程序上打上断点，使用密码模式重新获取token来理解流程，



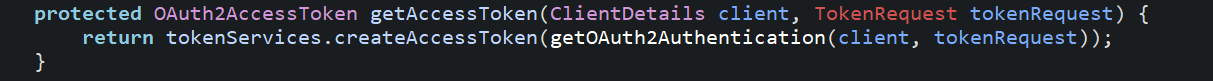
首先，TokenEndPoint第95行，进来后，第一步，从请求中获取client\_Id,并调用clientDetailsServices，获取第三方应用的详细配置，实际上，clientDetailsServices可以配置得很复杂，在本例中，只单独配置了appid和appsecret，这里已经拿到了clientDetails，之后，创建TokenRequest，之后，根据数据库中返回的client信息，和从前端传回来的信息进行对比，如果成功之后才进行后面的认证，之后，检测请求中的scope，实际上，在配置client的时候，就可以对scope进行配置，即，这个client可以发出什么样的授权，这里的校验逻辑是，第三方请求的scope一定要在他在服务商这边注册scope的范围之内，之后，再判断grant\_type参数的值，即，使用哪种方式来获取授权，之后，在判断grant\_type是否是简化模式，如果是简化模式也会抛出异常，在简化模式中不会需要访问token，再判断当前是否是授权码模式请求，因为在授权码模式里面，scope不是由这个获取token的请求决定的，而是在第一步获取授权码的时候，用户给与的授权码决定的，如果用户只给读权限，那么这个scope只能是读的权限，所以，如果是授权码认证请求，那么会直接将scope清空，后面会从授权码中获取权限，重新赋值，而不会用请求令牌请求中携带的scope，之后，判断当前获取token请求是否是刷新token请求，因为刷新令牌请求，也有自己约定好的cope，这里会对scope重新赋值，在经过一系列的检查之后，tokenRequest会传给TokenGranter，由他产生最终的AccessToken，写入到响应中。

在TokenGranter封装了4中授权，以及刷新令牌一共5中令牌生成的逻辑，进入该类中。

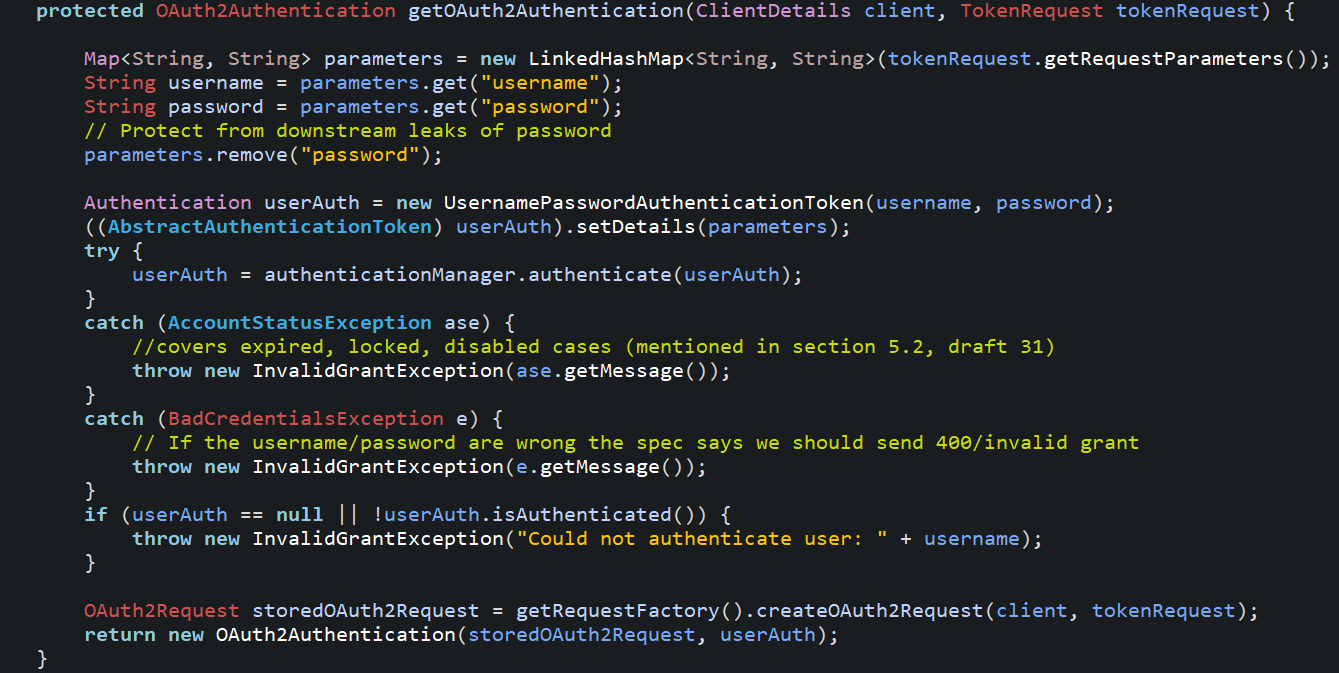
在debug模式中，可以观察到tokenGranters集合中的集合数量为5，这5个对象就对应着5种令牌生成方式，进入之后，很明显一个for循环，根据传入的grantType，从5个生成方式中挑一个生成最终的accessToken，最后，进入grant方法中，首先判断一下传入的请求类型与本类处理的逻辑是否相同，如果不同直接返回null。



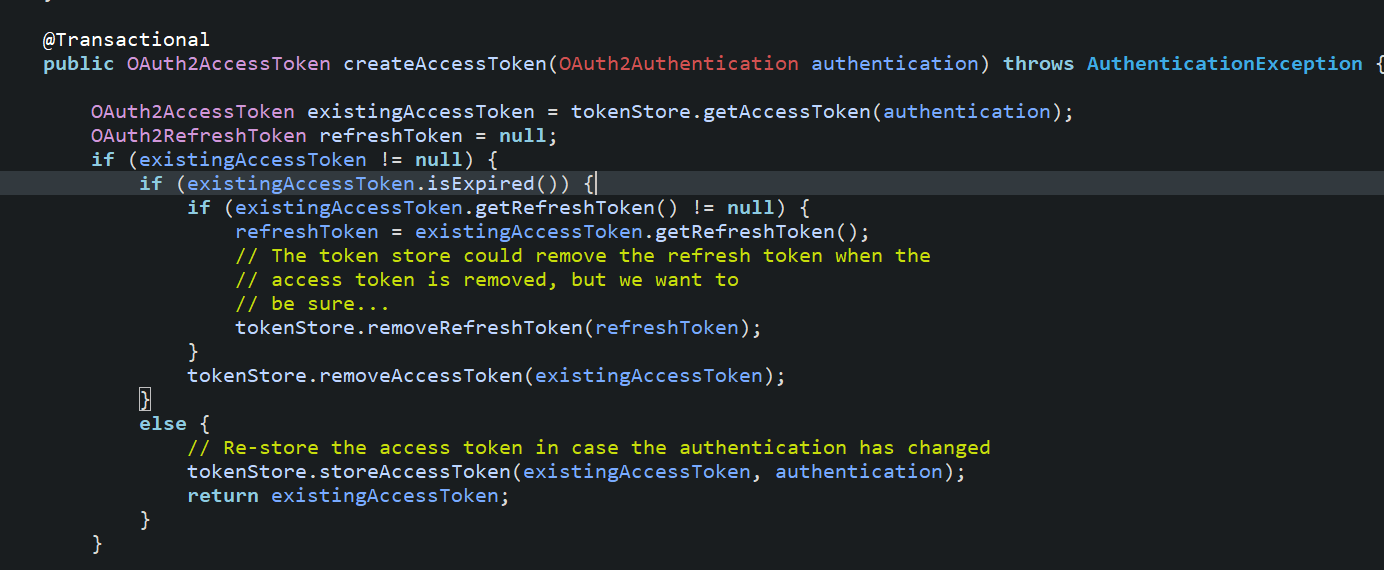
之后，在重新获取一次clientDetails，校验一次。最后，通过getAccessToken，产生最终的accessToken，在该方法里面，实际上就是在调用tokenServices,就是上图中最后一个业务逻辑类.



这里，针对不同的授权模式有不同的实现，ResourceOwnerPasswordTokenGranter类是针对于用户名密码授权模式token生成。



很明显，从请求中获取用户名密码，然后封装成UsernamePasswordAuthticationToken，之后交给authenticationManager去做认证，之后，拿到authentication之后，根据传入的tokenRequest和client包装成Oauth2Request，最终，在将Oauth2Request和authentication拼装为OAuth2Authentication，最后，如果有成功返回，那么整个token生成流程就结束了，通过响应返回到前端。之后，研究DefaultTokenServices，可以看到accessToken具体是如何生成的。



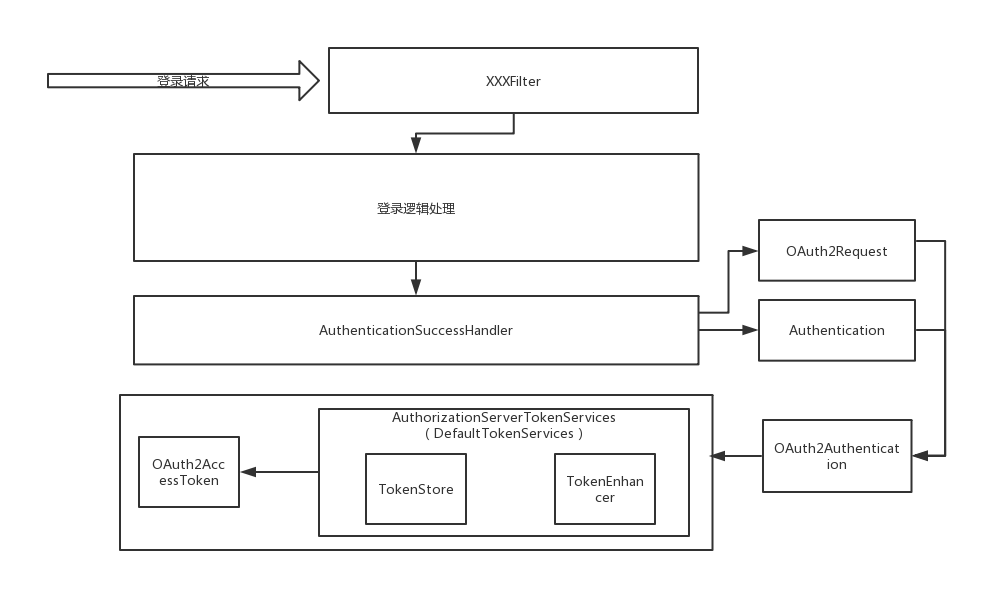
首先，通过tokenStore，从数据库中找寻当前用户是否已经有持久化的accessToken了，在之前的业务中也提到过，只要获取到了用户的token，无论用户是通过什么手段授权登录的，只要获取到了，那么token就会被持久化，下次再要获取token的时候，直接返回持久化的token，之后，如果已经存在的令牌不为空，判断令牌是否过期，那么从tokenStore中，将过期的令牌删除，如果没过期，会重新把令牌，存取一下，同时，将这个已经存在的令牌发挥。

如果这个令牌从数据库中没找到，那么则会新建一个令牌，拿authentication和refreshToken，创建Oauth2AccessToken，将，相应的成员变量传入，之后，调用tokenEnhancer，增强token，如果系统中有配置enhancer的话，之后，tokenstore存储token，最终把令牌返回。至此，整个token生成流程就完成了。

### 5-4.重构用户名密码登录

通过上小节的讲解，已经完成了整个token的生成流程，如果想要重构自己的用户名密码登录流程，那么，在tokenGranter之前的业务流程，基本都是不可更改的，即，接受生成token的请求，验证请求中的参数，根据授权模式的不同，调用不同的token生成器，这些逻辑很明显是不能覆盖的。那么，要如何才能根据业务需求，自定义token生成流程呢。

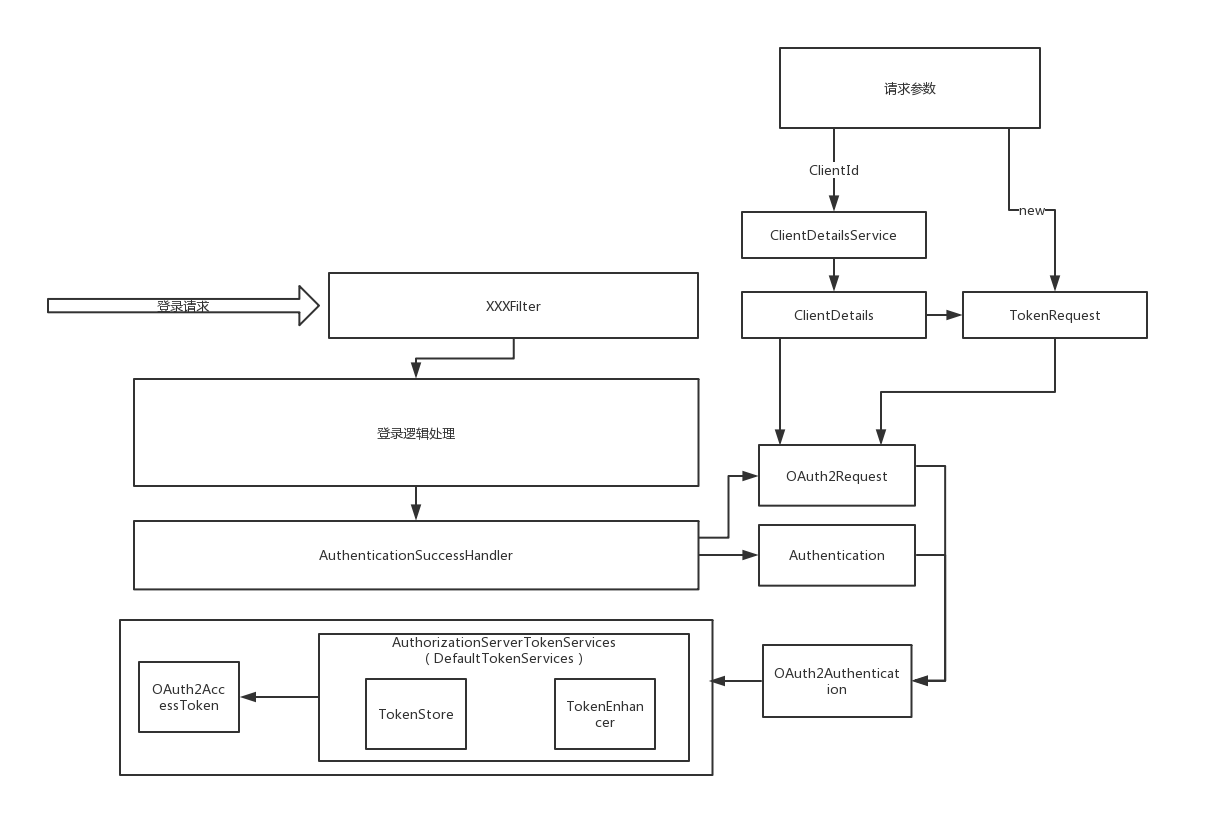
很简单，只需要在登录成功之后，生成token即可，即在authenticationSuccessHandler接口的实现内，去调用authorizationServerTokenServices让它生成token。调用该方法，需要使用authentication以及oauth2request拼装为oauth2Authentication。



在之前的授权模式中，是使用的请求中传入的信息，来构建OAuth2Authentication对象，而在，自定逻辑中，如果用户成功进行登录之后，会将authtication对象传入到handler中，在这里，不需要处理authentication对象，在这里只需要处理oauth2request，也就是说，在handler内，需要构建出oauth2request，那么则需要clientDetails，以及tokenRequest去拼装oauth2request，也就是说要调用clientDetailsServices，且需要一个clientID，这个参数需要从请求中获取，在请求中的base 64编码，将clientid以及clientsecret存储在请求中了。而tokenRequest是直接通过请求参数中的字段new出来的，那么整个流程就拼装完了。

实际上，主要就是需要拼装好oauth2authenticationToken对象，之后调用services，为了构建这个对象，需要在authenticationSuccessHandler中拼装逻辑。

那么，首先构建clientDetails，构建该对象需要得到请求中编码好的clientid以及clientsecret，首先需要解码，而这个解码逻辑，实际上在springsecurity 的base64登录中已经有了(baseAuthenticationFilter)，在这里，只需要将已经实现的代码复制过来并重构即可。



@Component

public class AuthenticationSuccessHandlerConfig extends SavedRequestAwareAuthenticationSuccessHandler {

    @Autowired

    private ObjectMapper objectMapper;

    @Autowired

    private ClientDetailsService clientDetailsService;

    @Autowired

    AuthorizationServerTokenServices authorizationServerTokenServices;

    /\* {"authorities":[{"authority":"admin"}],"details":{"remoteAddress":"0:0:0:0:0:0:0:1","sessionId":"166663DBDFE70D35AB332889AB41FF08"},"authenticated":true,"principal":{"password":null,"username":"mt","authorities":[{"authority":"admin"}],"accountNonExpired":true,"accountNonLocked":true,"credentialsNonExpired":true,"enabled":true},"credentials":null,"name":"mt"}

     \* (non-Javadoc)

     \* @see org.springframework.security.web.authentication.AuthenticationSuccessHandler#onAuthenticationSuccess(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, org.springframework.security.core.Authentication)

     \*/

    @Override

    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,

            Authentication authentication) throws IOException, ServletException {

        // TODO Auto-generated method stub

        String header = request.getHeader("Authorization");

        if (header == null || !header.startsWith("Basic ")) {

            throw new UnapprovedClientAuthenticationException("请求中无client信息");

        }

        String[] tokens = extractAndDecodeHeader(header, request);

        assert tokens.length == 2;

        // 这里的username password就是username，password

        String clientId = tokens[0];

        String clientSecret = tokens[1];

        ClientDetails clientDetail = clientDetailsService.loadClientByClientId(clientId);

        if (clientDetail == null) {

            throw new UnapprovedClientAuthenticationException("client Id 对应的信息再服务商中不存在" + clientId);

        }

        if (!StringUtils.equals(clientSecret, clientDetail.getClientSecret())) {

            throw new UnapprovedClientAuthenticationException("client Secret 信息不匹配" + clientId);

        }

        // 实际上就是在调用默认的token生成工具，只不过这里的生成凭证是用户登录成功之后服务器自己拼装的

        // 在这里，不做scope校验,因为本系统是在公司内部使用，就返回所有授权即可，暂时不需要做权限授予 之后，授权类型是custom，即自定义授权方式。

        @SuppressWarnings("unchecked")

        TokenRequest tokentRequest = new TokenRequest(MapUtils.EMPTY\_SORTED\_MAP, clientId, clientDetail.getScope(),

                "custom");

        OAuth2Request auth2Request = tokentRequest.createOAuth2Request(clientDetail);

        OAuth2Authentication oAuth2Authentication = new OAuth2Authentication(auth2Request, authentication);

        OAuth2AccessToken accessToken = authorizationServerTokenServices.createAccessToken(oAuth2Authentication);

        response.setContentType("application/json;charset=utf-8");

        response.getWriter().write(objectMapper.writeValueAsString(accessToken));

    }

    /\*\*

     \*

    \* <b>Description:收到请求头之后，首先将请求头解码，会得到

    \* usernmae:password

    \* 这样的字符串，然后找到冒号的位置，如果没找到冒号，则解码错误

    \* 找到冒号的位置时候，构建新的字符数组

    \* 字符串开始到冒号为第一个，冒号开始到字符串结尾为第二个

    \* 即

    \* token[0]=username

    \* token[1]=password

    \* </b><br>

    \* @param header

    \* @param request

    \* @return

    \* @throws IOException

    \* @Note

    \* <b>Author:</b> <a href="" target="\_blank">Monty</a>

    \* <br><b>Date:</b> 2019年3月15日 上午11:27:16

    \* <br><b>Version:</b> 1.0

     \*/

    private String[] extractAndDecodeHeader(String header, HttpServletRequest request) throws IOException {

        byte[] base64Token = header.substring(6).getBytes("UTF-8");

        byte[] decoded;

        try {

            decoded = Base64.decode(base64Token);

        } catch (IllegalArgumentException e) {

            throw new BadCredentialsException("Failed to decode basic authentication token");

        }

        String token = new String(decoded, "UTF-8");

        int delim = token.indexOf(":");

        if (delim == -1) {

            throw new BadCredentialsException("Invalid basic authentication token");

        }

        return new String[] { token.substring(0, delim), token.substring(delim + 1) };

    }

}

至此，当用户登录之后，由登录过滤器验证用户信息是否正确，之后，当过滤器认证通过之后，获得authentication，交由successhandler继续处理，调用系统存在的accessToken生成类，生成token之后，以json形式写入响应中，返回给前端。

在这之后，在对整个app认证服务器加上安全配置。

@Configuration

@EnableResourceServer

public class SeehopeResourceServerConfig extends ResourceServerConfigurerAdapter {

    @Autowired

    private SmsCodeAuthenticationSecurityConfig smsCodeAuthenticationSecurityConfig;

    @Autowired

    private SpringSocialConfigurer seehopeSocialConfigurer;

    @Autowired

    private FormAuthenticationConfig formAuthenticationConfig;

    @Autowired

    private AuthorizeConfigManager authorizeConfigManager;

    /\* (non-Javadoc)

     \* @see org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerConfigurerAdapter#configure(org.springframework.security.config.annotation.web.builders.HttpSecurity)

     \*/

    @Override

    public void configure(HttpSecurity http) throws Exception {

        // TODO Auto-generated method stub

        formAuthenticationConfig.configure(http);

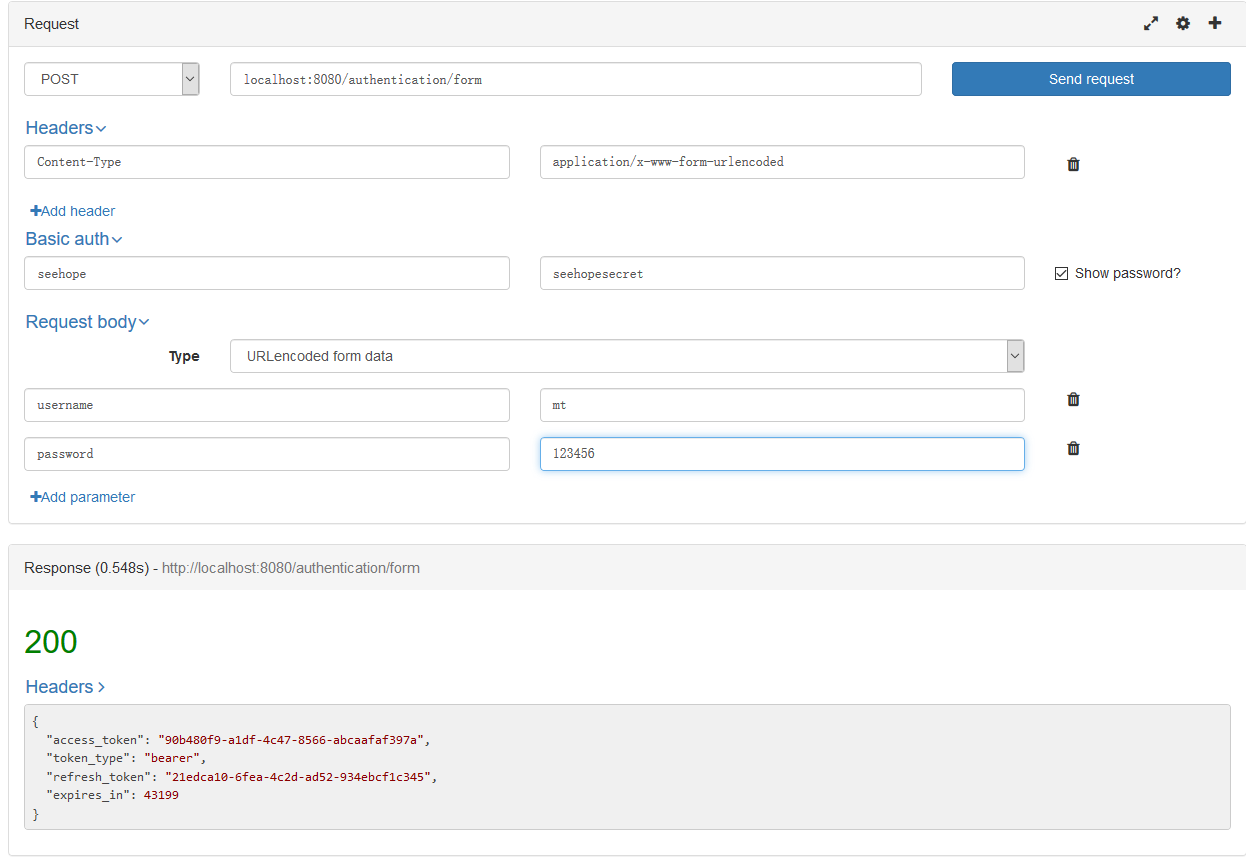
        http.apply(smsCodeAuthenticationSecurityConfig).and().apply(seehopeSocialConfigurer).and().csrf().disable();

        authorizeConfigManager.config(http.authorizeRequests());

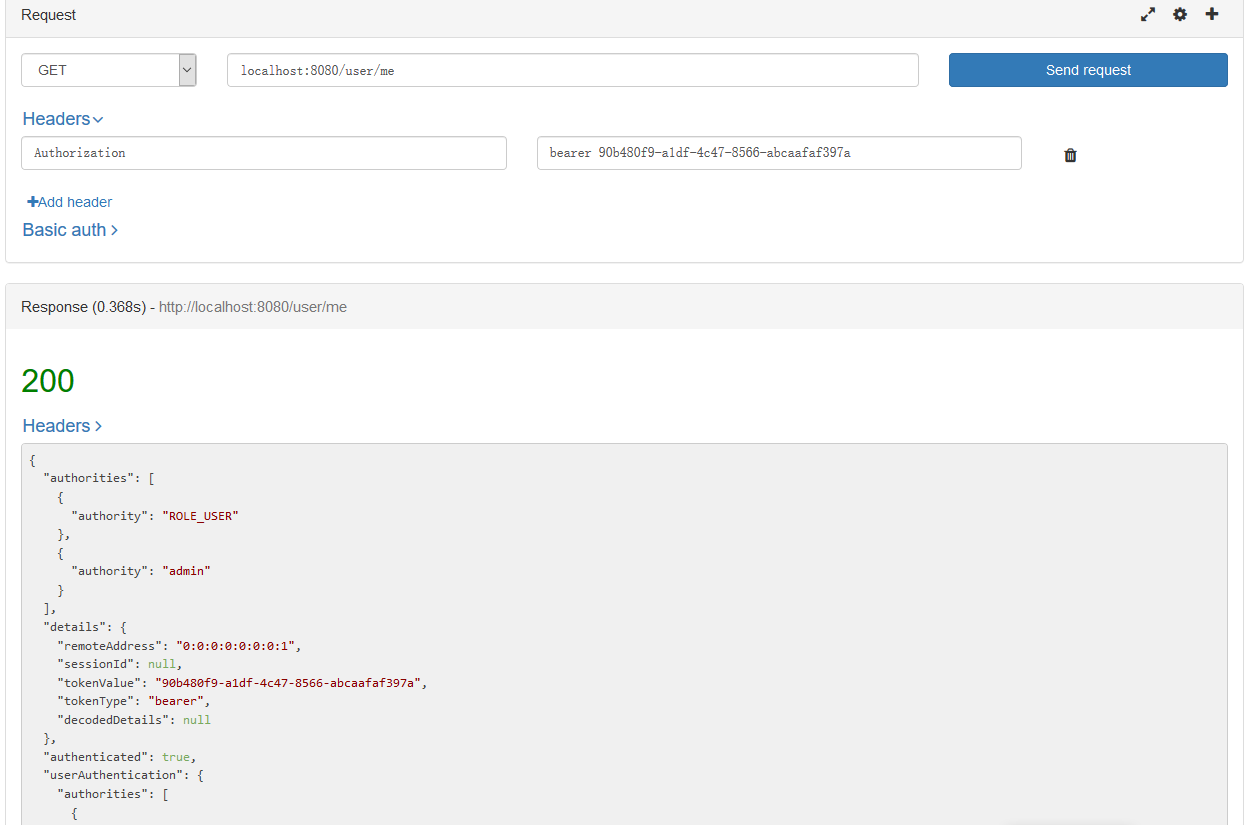
    }

}

这里的安全配置基本上和browser中的配置相同，只不过这里的登陆成功处理器逻辑被重构了，之后，再次通过工具访问登录请求/authentication/form，携带用户名密码，何在请求头中的authorization信息。



在这里，成功的获取到了token。



之后，再访问资源的时候，将token携带上，资源服务器返回了终端。

### 5-5.重构短信登录

### 5-6.重构社交登录

### 5-7.重构注册逻辑

### 5-8.令牌配置

### 5-9.使用JWT替换默认令牌

### 5-10.基于JWT实现SSO单点登录