OAuth2 Knowledge Graph

该文章旨在说明OAuth2实现与改造方面做个哪些事情，会从以下几个方面来说明:

基础篇

对OAuth2的了解可以参考,
* 理解OAuth 2.0
* OAuth2 官方网站
* OAuth2 Spec

授权模式(grant_type)

授权模式	符号代码	支持refresh token	备注
密码模式	password	√	为遗留系统设计
授权码模式	authorization_code	√	标准授权方式
简化模式	implicit	x	为web浏览器应用设计
客户端模式	client_credentials	x	为后台api服务消费者设计

授权类型(reponse_type)

符号代码	对应授权模式	备注
token	简化模式	客户端在浏览器请求authorize的url，需带上response_
code	授权码模式	客户端在浏览器请求authorize的url，需带上response_

接口篇

接口列表

序号	url	method	备注

1	/oauth/authorize	GET	authenticate成功后，会回跳的授权url
2	/oauth/authorize	POST	approval的form表单请求
3	/oauth/token	GET	如未设置允许GET请求，调用该请求会抛 supported，否则回跳POST /oauth/toke
4	/oauth/token	POST	生成token的入口
5	/oauth/check_token	GET	校验token的有效性，并展示token包含的
6	/oauth/confirm_access	GET	在执行GET /oauth/authorize后，如需要确认approv
7	/oauth/error	GET	OAuth2 内部错误时，会以该url形式以页
8	/oauth/token_key	POST	JWT encode时，可以找到其Key值
9	/oauth/login	POST	授权码模式form表单登录，在filter里配

扩展篇

跳转到外面的登录页面

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.addFilterBefore(jsonFilter, ChannelProcessingFilter.class);
  http.authorizeRequests().
      ...
      .and().formLogin()
      .loginPage(loginPage)
      ...
}
```

指定该loginPage值为 https://xxx.dev.yy.com/oauth/login

跳转的登录页附带额外的参数

比如 https://xxx.dev.yy.com/oauth/login?partner_key=xxx

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.addFilterBefore(jsonFilter, ChannelProcessingFilter.class);
  http.authorizeRequests().
```

```
        ...
        .and()
        .exceptionHandling()
        .authenticationEntryPoint( //
            new CustomLoginUrlAuthenticationEntryPoint(loginPage));
        ...
}
```

其中 CustomLoginUrlAuthenticationEntryPoint 实现了 AuthenticationEntryPoint接口，在buildRedirectUrlToLoginPage方法里可以进行设置。

```
if (UrlUtils.isAbsoluteUrl(loginForm)) {
  // Modify by tao.yang 2018/07/26
  return loginForm + "?" + REQUEST_PARTNER_KEY + "=" + request.getParam
eter(PARTNER_KEY) + //
      "&" + REQUEST_LOGIN_TYPE + "=" + request.getParameter(LOGIN_TYPE)
 + //
      "&" + DISPLAY + "=" + request.getParameter(DISPLAY);
}
...
// Add by tao.yang 2018/07/26
urlBuilder.setQuery(REQUEST_PARTNER_KEY + "=" + request.getParameter(PA
RTNER_KEY));
urlBuilder.setQuery(REQUEST_LOGIN_TYPE + "=" + request.getParameter(LOG
IN_TYPE));
urlBuilder.setQuery(DISPLAY + "=" + request.getParameter(DISPLAY));
```

跳转到外面的确认授权页面

原生的GET /oauth/authorize接口，实现的是AuthorizationEndpoint.authorize中，

```
private ModelAndView getUserApprovalPageResponse(Map<String, Object> mo
del,
  AuthorizationRequest authorizationRequest, Authentication principal)
{
 ...
 return new ModelAndView(userApprovalPage, model);
}
```

直接跳转到内置的/oauth/confirm_access处。

需要修改成，已重写在AuthorizationController.authorize里

```
private ModelAndView getUserApprovalPageResponse(Map<String, Object> mo
```

```
del,
    AuthorizationRequest authorizationRequest, Authentication principal
) {
  ...
  return new ModelAndView(new RedirectView(userApprovalPage), model);
}
```

代码里的userApprovalPage配置为 https://xx.dev.yy.com/oauth/confirm_access
直接redirect出去。

实现AuthenticationProvider提供业务登录验证

```
@Autowired
private CustomAuthenticationProvider authenticationProvider;
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exce
ption {
  auth.authenticationProvider(authenticationProvider);
}
```

登录成功后，回跳/oauth/authorize接口，需要再次从API网关进入，需要修改域名

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.addFilterBefore(jsonFilter, ChannelProcessingFilter.class);
  http.authorizeRequests().
      ...
      .and().formLogin()
      .successHandler(new CustomAuthenticationSuccessHandler(osgServerH
ost))
      ...
}
```

参数osgServerHost是可配置的，目前都是配置成https。

给OAuth2AccessToken增加附加信息

```
@Bean
public TokenEnhancer tokenEnhancer() {
  return new CustomTokenEnhancer();
}
```

```
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints)
 throws Exception {
  endpoints.authenticationManager(authenticationManager);
  ...
  TokenEnhancerChain tokenEnhancerChain = new TokenEnhancerChain();
  tokenEnhancerChain.setTokenEnhancers(Arrays.asList(tokenEnhancer(), a
ccessTokenConverter()));
  endpoints.tokenEnhancer(tokenEnhancerChain);
  ...
}


@Bean
public JwtAccessTokenConverter accessTokenConverter() {
  CustomJwtAccessTokenConverter converter = new CustomJwtAccessTokenCon
verter();
  ...
  return converter;
}
```

CustomTokenEnhancer实现了TokenEnhancer接口，我们在enhance方法中从userAuthentication拿到details数据进行填充。

JWT token refresh的时候如何带上附加信息

和上一条类似，需要在CustomTokenEnhancer.enhance中设置，

```
// refresh token need set OAuth2Authentication.getDetails
// Add by tao.yang @since 2018/08/14
Object detailsObject = authentication.getDetails();
if (detailsObject != null && detailsObject instanceof Map) {
  additionalInfo.putAll((Map)detailsObject);
}
```

还有，重写的CustomTokenService.createRefreshedAuthentication中设置将details写出来。

```
narrowed = new OAuth2Authentication(clientAuth, authentication.getUserA
uthentication());
// Add by tao.yang @Since 2018/08/14
if (authentication.getDetails() != null) {
  narrowed.setDetails(authentication.getDetails());
}
return narrowed;
```

实现篇

客户端模式

请求方式

```
POST /oauth/token HTTP/1.1
Host: localhost:9080
Content-Type: application/json
Cache-Control: no-cache
Postman-Token: f4dff8d2-3794-32a1-cd97-3c03ea8fd671

{
    "clientId": "client",
    "clientSecret": "secret",
    "grantType": "client_credentials"
}
```

执行过程的filter list

序号	名称	备注
1	WebAsyncManagerIntegrationFilter	
2	SecurityContextPersistenceFilter	
3	HeaderWriterFilter	
4	LogoutFilter	
5	ClientCredentialsTokenEndpointFilter	*
6	BasicAuthenticationFilter	
7	RequestCacheAwareFilter	
8	SecurityContextHolderAwareRequestFilter	
9	AnonymousAuthenticationFilter	
10	SessionManagementFilter	
11	ExceptionTranslationFilter	

12	FilterSecurityInterceptor	

ClientCredentialsTokenEndpointFilter 在这些filter里比较重要，会在attemptAuthentication去验证请求method以及client_id和client_secret信息，

```
public Authentication attemptAuthentication(HttpServletRequest request,
 HttpServletResponse response) throws AuthenticationException, IOExcept
ion, ServletException {
  if (this.allowOnlyPost && !"POST".equalsIgnoreCase(request.getMethod(
))) {
    throw new HttpRequestMethodNotSupportedException(request.getMethod(
), new String[]{"POST"});
  } else {
    String clientId = request.getParameter("client_id");
    String clientSecret = request.getParameter("client_secret");
    Authentication authentication = SecurityContextHolder.getContext().
getAuthentication();
    if (authentication != null && authentication.isAuthenticated()) {
      return authentication;
    } else if (clientId == null) {
      throw new BadCredentialsException("No client credentials presente
d");
    } else {
      if (clientSecret == null) {
        clientSecret = "";
      }

      clientId = clientId.trim();
      UsernamePasswordAuthenticationToken authRequest = new UsernamePas
swordAuthenticationToken(clientId, clientSecret);
      return this.getAuthenticationManager().authenticate(authRequest);
    }
  }
}
```

代码中getAuthenticationManager()返回的对象是DaoAuthenticationProvider，会去执行AbstractUserDetailsAuthenticationProvider.authenticate去检索client details信息。

```
try {
 loadedUser = this.getUserDetailsService().loadUserByUsername(username)
;
}
```

以上这些执行完成后，回去真正请求POST /oauth/token会去做一些必要的checked，关键点在生成token，具体在ClientCredentialsTokenGranter.grant方法中，

```
public OAuth2AccessToken grant(String grantType, TokenRequest tokenRequ
est) {
  OAuth2AccessToken token = super.grant(grantType, tokenRequest);
  ...
}
```

最后会去调用tokenService.createAccessToken

密码模式

请求方式,

```
POST /oauth/token HTTP/1.1
Host: localhost:9080
Content-Type: application/json
X-BK-Data-Encrypt-Decrypt: false
Cache-Control: no-cache
Postman-Token: e75e8595-9c5c-708e-6c7e-237ca7c77538

{
    "accountName": "15711341472",
    "password": "883876li",
    "authType": "PASSWORD",
    "accountType": "PHONE",
    "deviceInfo": {
        "appIP":"127.0.0.1"
    },
    "partnerKey": "***",
    "display": "default",
    "grantType": "password",
    "clientId": "ci-0823",
    "clientSecret": "MrQx28VIOfCtFQXjf4teDqQFMntrjjEp"
}
```

执行过程中的filter list

序号	名称	备注
1	WebAsyncManagerIntegrationFilter	
2	SecurityContextPersistenceFilter	
3	HeaderWriterFilter	
4	LogoutFilter	

5	UsernamePasswordAuthenticationFilter	*
6	RequestCacheAwareFilter	
7	SecurityContextHolderAwareRequestFilter	
8	AnonymousAuthenticationFilter	
9	SessionManagementFilter	
10	ExceptionTranslationFilter	
11	FilterSecurityInterceptor	

对比客户端模式的filters少了个BasicAuthenticationFilter，且没有了ClientCredentialsTokenEndpointFilter换成了UsernamePasswordAuthenticationFilter了。

咱们先来看看UsernamePasswordAuthenticationFilter, 经过该filter的条件是

```
public UsernamePasswordAuthenticationFilter() {
 super(new AntPathRequestMatcher("/login", "POST"));
}
```

真正执行接下来的操作的方法是 attemptAuthentication

```
public Authentication attemptAuthentication(HttpServletRequest request,
  HttpServletResponse response) throws AuthenticationException {
 if (postOnly && !request.getMethod().equals("POST")) {
  throw new AuthenticationServiceException(
    "Authentication method not supported: " + request.getMethod());
 }

 String username = obtainUsername(request);
 String password = obtainPassword(request);

 if (username == null) {
  username = "";
 }

 if (password == null) {
  password = "";
 }

 username = username.trim();

 UsernamePasswordAuthenticationToken authRequest = new UsernamePassword
```

```
AuthenticationToken(
   username, password);

 // Allow subclasses to set the "details" property
 setDetails(request, authRequest);

 return this.getAuthenticationManager().authenticate(authRequest);
}
```

代码中getAuthenticationManager()返回的对象是CustomAuthenticationProvider，会去执行authenticate去调用uus的登录接口。

```
@Override
public Authentication authenticate(Authentication authentication) throw
s AuthenticationException {
  ...
  // invoke client to real login
  Map<String, Object> responsesDetails = new HashMap<>();
  responsesDetails.put(PARTNER_KEY, requestBody.get(REQUEST_UUS_PARTNER
_KEY)); // @see CustomApprovalStoreUserApprovalHandler
  if (requestBody.get(LOGIN_TYPE).equals(TERMINAL_LOGIN_TYPE_B)) {
    // B端 商户登录
    requestBody.remove(LOGIN_TYPE); // must be
    processBTerminalLogin(requestBody, responsesDetails);
  }
  ...
  UsernamePasswordAuthenticationToken usernamePasswordAuthenticationTok
en = new UsernamePasswordAuthenticationToken(
      principal, password, grantedAuthority);
  ...
  return usernamePasswordAuthenticationToken;
}
```

以上这些执行完成后，回去真正请求POST /oauth/token会去做一些必要的checked，关键点在生成token，具体在ResourceOwnerPasswordTokenGranter.grant方法中，

最后还是需要去调用tokenService.createAccessToken 其中要先去构造OAuth2Authentication，代码细节在ResourceOwnerPasswordTokenGranter.getOAuth2Authentication

```
protected OAuth2Authentication getOAuth2Authentication(ClientDetails cl
ient, TokenRequest tokenRequest) {

 Map<String, String> parameters = new LinkedHashMap<String, String>(tok
enRequest.getRequestParameters());
 String username = parameters.get("username");
```

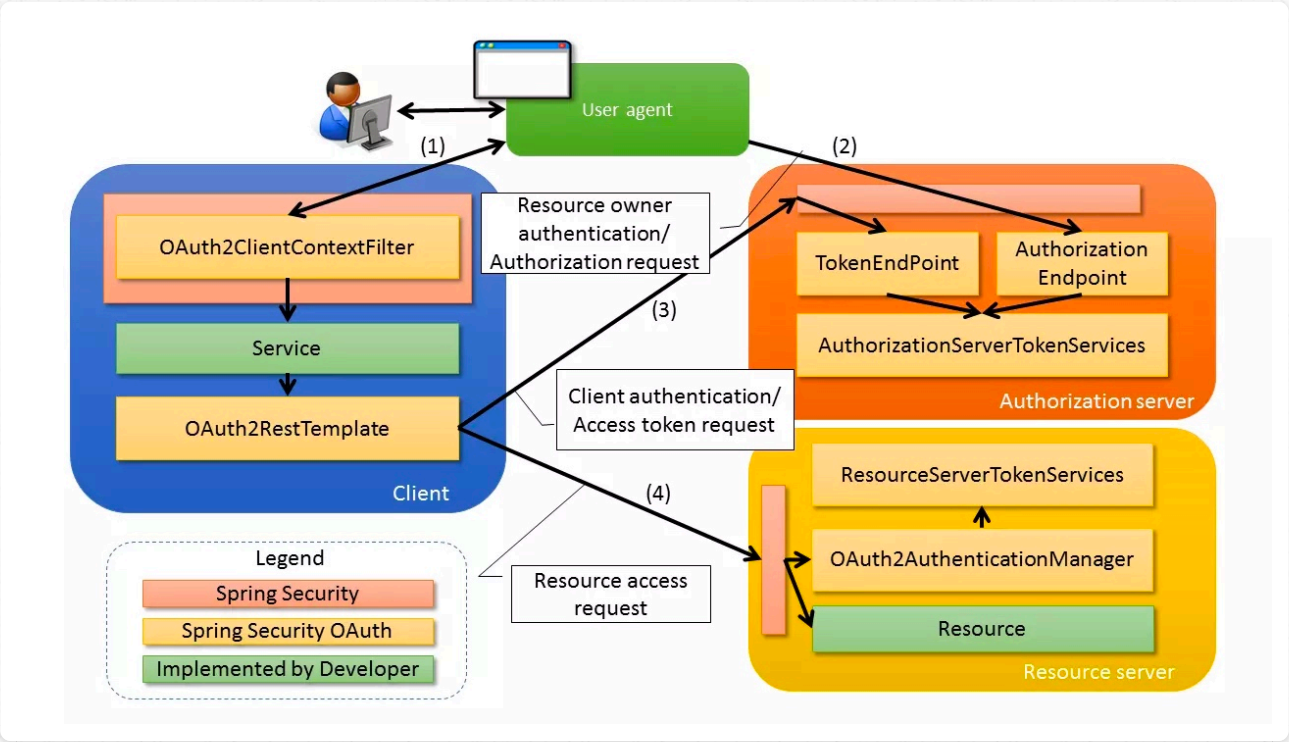
```
 String password = parameters.get("password");
 // Protect from downstream leaks of password
 parameters.remove("password");

 Authentication userAuth = new UsernamePasswordAuthenticationToken(user
name, password);
 ((AbstractAuthenticationToken) userAuth).setDetails(parameters);
 try {
  userAuth = authenticationManager.authenticate(userAuth);
 }
 catch (AccountStatusException ase) {
  //covers expired, locked, disabled cases (mentioned in section 5.2, d
raft 31)
  throw new InvalidGrantException(ase.getMessage());
 }
 catch (BadCredentialsException e) {
  // If the username/password are wrong the spec says we should send 40
0/invalid grant
  throw new InvalidGrantException(e.getMessage());
 }
 if (userAuth == null || !userAuth.isAuthenticated()) {
  throw new InvalidGrantException("Could not authenticate user: " + use
rname);
 }

 OAuth2Request storedOAuth2Request = getRequestFactory().createOAuth2Re
quest(client, tokenRequest);
 return new OAuth2Authentication(storedOAuth2Request, userAuth);
}
```

架构篇

Spring Security OAuth2架构



- 资源拥有者通过UserAgent访问client，在授权允许访问授权端点的情况下，OAuth2RestTemplate会创建OAuth2认证的REST请求，指示UserAgent重定向到Authorization Server的授权端点AuthorizationEndpoint。
- UserAgent访问Authorization Server的授权端点的authorize方法，当未注册授权时，授权端点将需要授权的界面/oauth/confirm_access显示给资源拥有者，资源拥有者授权后会通过 AuthorizationServerTokenServices生成授权码或访问令牌，生成的令牌最终会通过userAgent重定向传递给客户端。
- 客户端的OAuth2RestTemplate拿到授权码后创建请求访问授权服务器TokenEndpoint令牌端点，令牌端点通过调用AuthorizationServerTokenServices来验证客户端提供的授权码进行授权，并颁发访问令牌响应给客户端。
- 客户端的OAuth2RestTemplate在请求头中加入从授权服务器获取的访问令牌来访问资源服务器，资源服务器通过OAuth2AuthenticationManager调用ResourceServerTokenServices验证访问令牌和与访问令牌关联的验证信息。访问令牌验证成功后，返回客户端请求对应的资源。

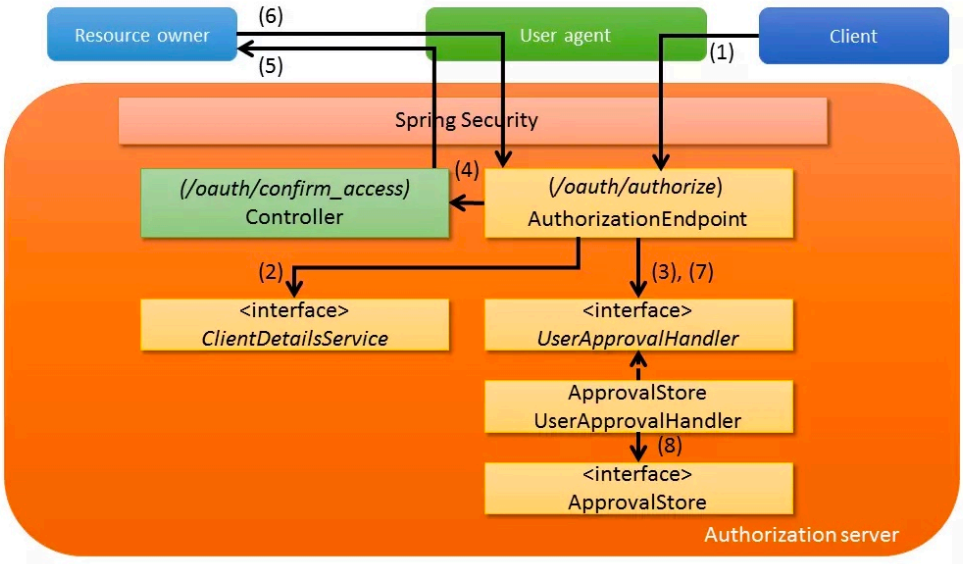
Authorization Server（授权服务器）架构

授权服务器主要提供了资源拥有者的认证服务，客户端通过授权服务器向资源拥有者获取授权，然后获取授权服务器颁发的令牌。

在这个认证流程中，涉及到两个重要端点，一个是授权端点AuthorizationEndpoint，另一个是令牌端点TokenEndpoint。

AuthorizationEndpoint（授权端点）

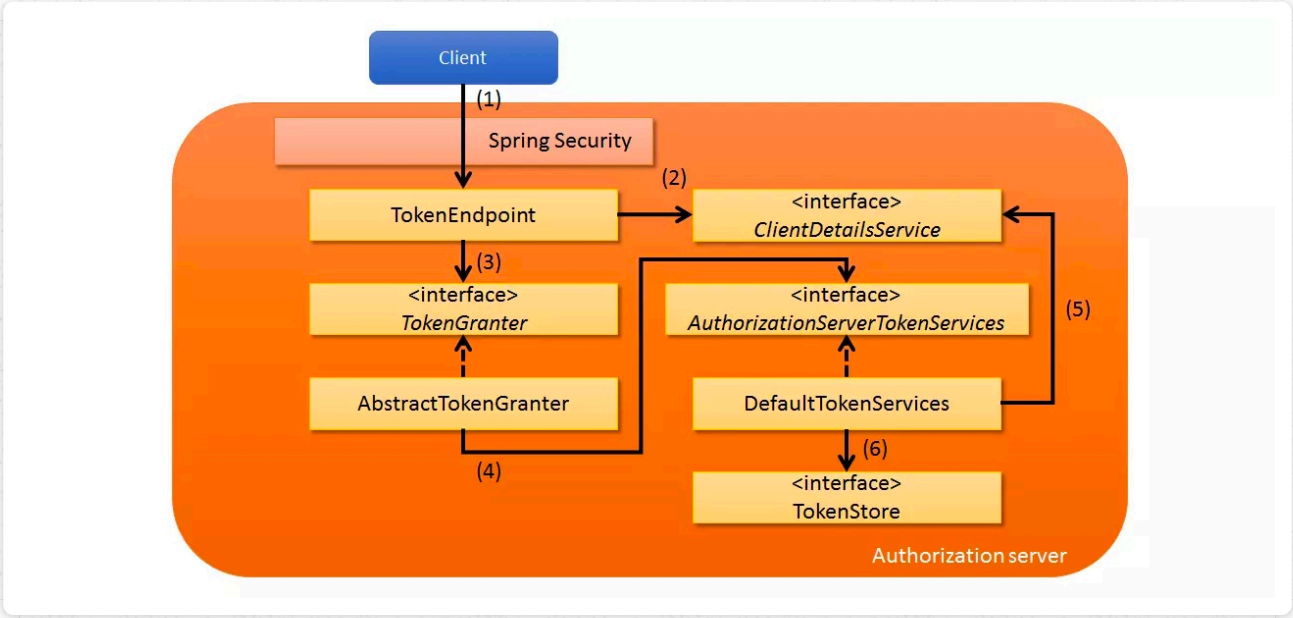
首先让我们来看下访问授权端点AuthorizationEndpoint的执行流程：



- UserAgent会访问授权服务器的AuthorizationEndpoint（授权端点）的URI：/oauth/authorize，调用的是authorize方法，主要用于判断用户是否已经授权，如果授权颁发新的authorization_code，否则跳转到用户授权页面。
- authorize它会先调用ClientDetailsService获取客户端详情信息，并验证请求参数。随后authorize方法再将请求参数传递给UserApprovalHandler用来检测客户端是否已经注册了scope授权。
- 当未注册授权时，即approved为false，将会向资源拥有者显示请求授权的界面/oauth/confirm_access。 同4一致。
- 资源拥有者确认授权后会再次访问授权服务器的授权端点的URI：/oauth/authorize，此次请求参数会增加一个user_oauth_approval，因此会调用另一个映射方法approveOrDeny。
- approveOrDeny会调用userApprovalHandler.updateAfterApproval根据用户是否授权，来决定是否更新authorizationRequest对象中的approved属性。
- userApprovalHandler的默认实现类是ApprovalStoreUserApprovalHandler，其内部是通过ApprovalStore的addApprovals来注册授权信息的。

TokenEndpoint（令牌端点）

接下来我们看下令牌端点TokenEndpoint的执行流程：



- userAgent通过访问授权服务器令牌端点TokenEndpoint的URI：/oauth/token，调用的是postAccessToken方法，主要用于为客户端生成Token。
- postAccessToken首先会调用ClientDetailsService获取客户端详情信息并验证请求参数。
- 调用对应的授权模式实现类生成Token。
- 对应的授权模式都是实现了AbstractTokenGranter抽象类，它的成员AuthorizationServerTokenServices可以用来创建、刷新、获取token。
- AuthorizationServerTokenServices默认实现类只有DefaultTokenServices，通过它的createAccessToken方法可以看到token是如何创建的。
- 真正操作token的类是TokenStore，程序根据TokenStore接口的不同实现来生产和存储token。

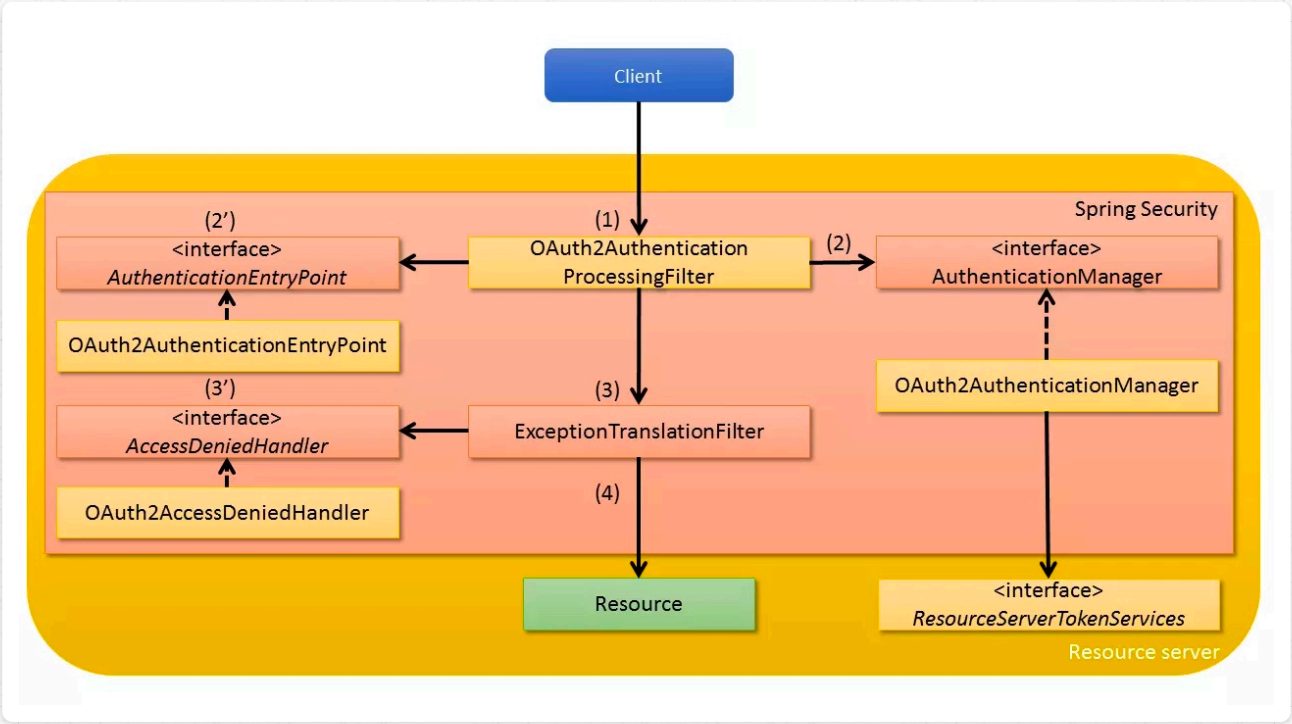
Resource Server（资源服务器）架构

资源服务器主要用于处理客户端对受保护资源的访问请求并返回相应。

资源服务器会验证客户端的访问令牌是否有效，并获取与访问令牌关联的认证信息。

获取认证信息后，验证访问令牌是否在允许的scope内，验证完成后的处理行为可以类似于普通应用程序来实现。

下面是资源服务器的运行流程：



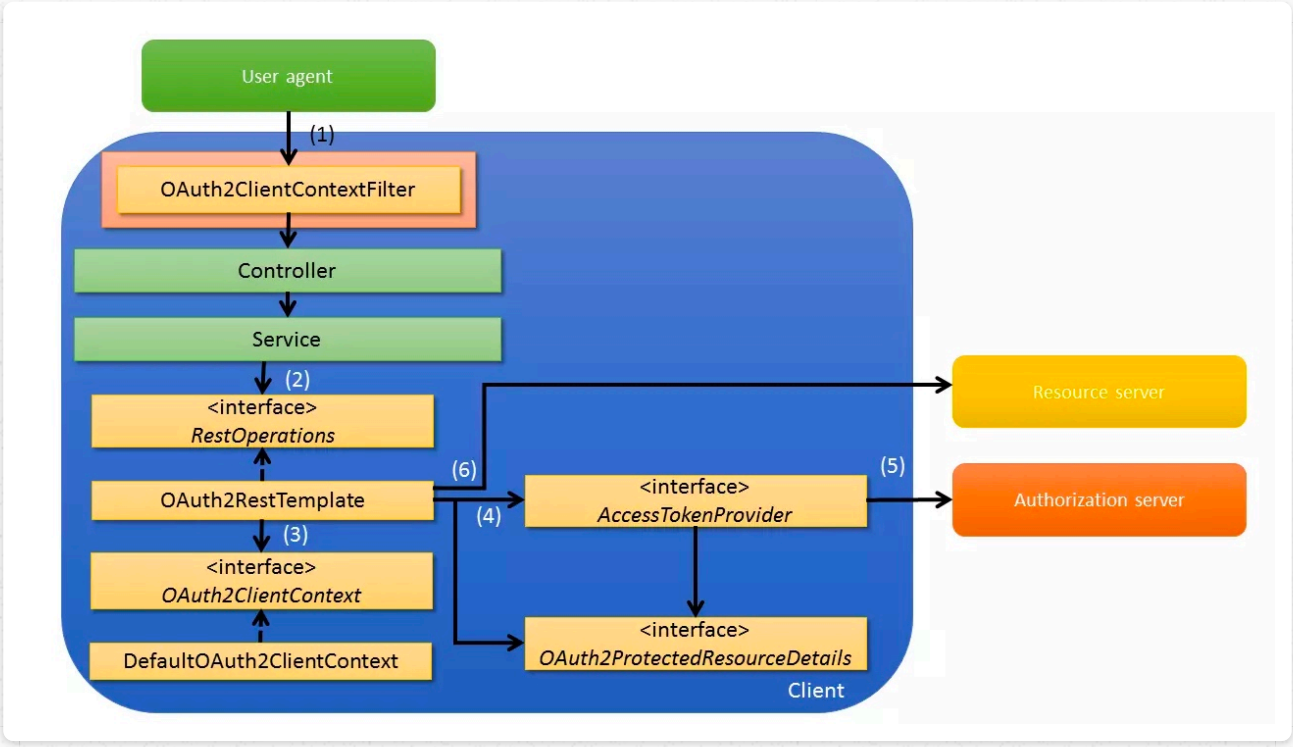
- 客户端开始访问资源服务器时，会先经过OAuth2AuthenticationProcessingFilter，这个拦截器的作用是从请求中提取访问令牌，然后从令牌中提取认证信息Authentication并将其存放到上下文中。
- OAuth2AuthenticationProcessingFilter拦截器中会调用AuthenticationManager的authenticate方法提取认证信息。
- OAuth2AuthenticationProcessingFilter拦截器如果发生认证错误时，将委托AuthenticationEntryPoint做出错误响应，默认实现类是OAuth2AuthenticationEntryPoint。
- OAuth2AuthenticationProcessingFilter执行完成后进入下一个安全过滤器ExceptionTranslationFilter。
- ExceptionTranslationFilter过滤器用来处理在系统认证授权过程中抛出的异常，拦截器如果发生异常，将委托AccessDeniedHandler做出错误响应，默认实现类是OAuth2AccessDeniedHandler。
- 当请求的认证/授权验证成功后，返回客户得请求对应的资源

Client（客户端）架构

Spring security OAuth2客户端控制着OAuth 2.0保护的其它服务器的资源的访问权限。

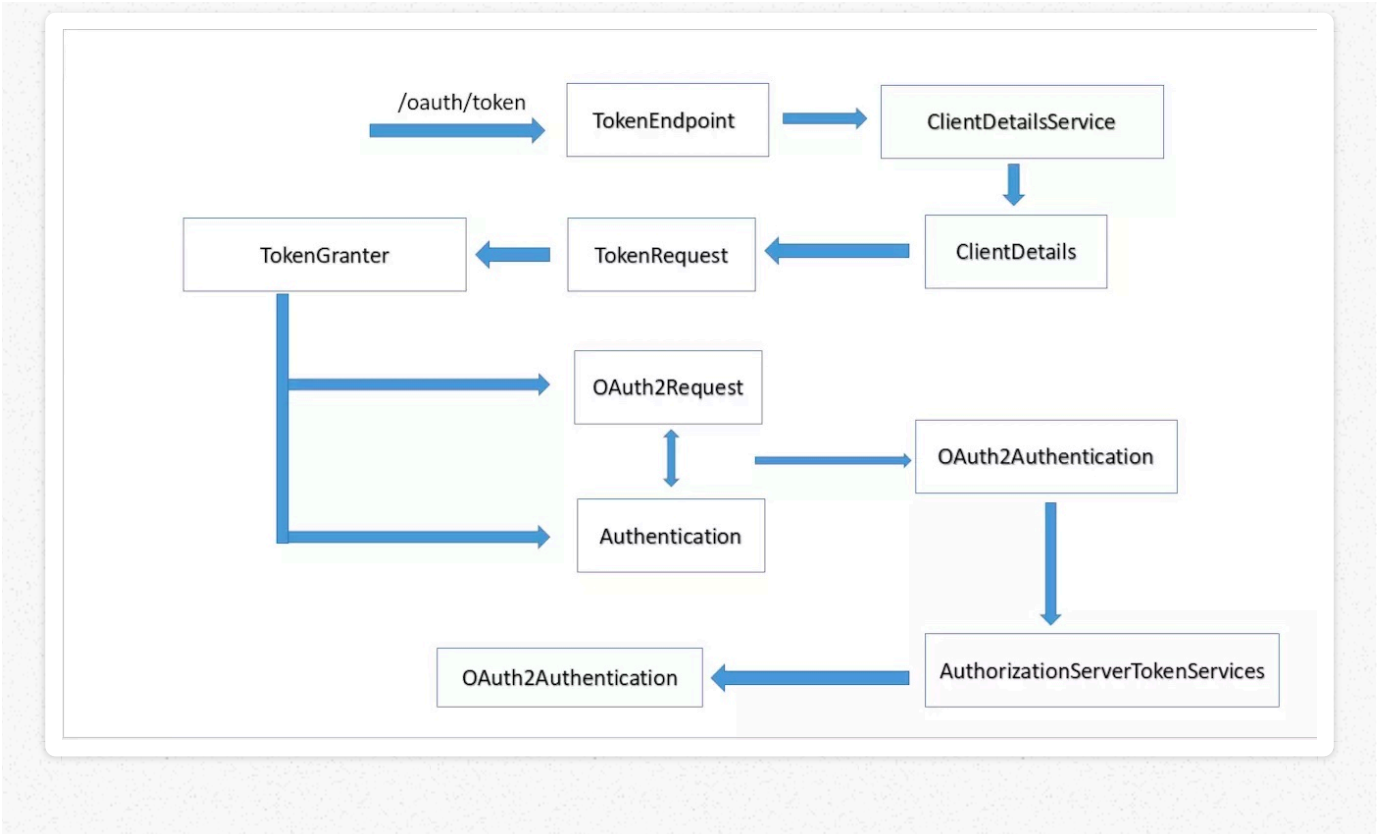
配置包括建立相关受保护资源与有权限访问资源的用户之间的连接。

客户端也需要实现存储用户的授权代码和访问令牌的功能。



- 首先UserAgent调用客户端的Controller，在这之前会经过OAuth2ClientContextFilter过滤器，它主要用来捕获第5步可能发生的UserRedirectRequiredException，以便重定向到授权服务器重新授权。
- 客户端service层相关代码需要注入RestOperations->OAuth2RestOperations接口的实现类OAuth2RestTemplate。它主要提供访问授权服务器或资源服务器的RestAPI。
- OAuth2RestTemplate的成员OAuth2ClientContext接口实现类为DefaultOAuth2ClientContext。它会校验访问令牌是否有效，有效则执行第6步访问资源服务器。
- 如果访问令牌不存在或者超过了有效期，则调用AccessTokenProvider来获取访问令牌。
- AccessTokenProvider根据定义的资源详情信息和授权类型获取访问令牌，如果获取不到，抛出UserRedirectRequiredException。
- 指定3或5中获取的访问令牌来访问资源服务器。如果在访问过程中发生令牌过期异常，则初始化所保存的访问令牌，然后走第4步。

核心以及类图


/oauth/token
TokenEndpoint
ClientDetailsService
TokenGranter
TokenRequest
ClientDetails
OAuth2Request
OAuth2Authentication
Authentication
OAuth2Authentication
AuthorizationServerTokenServices