# Oauth2

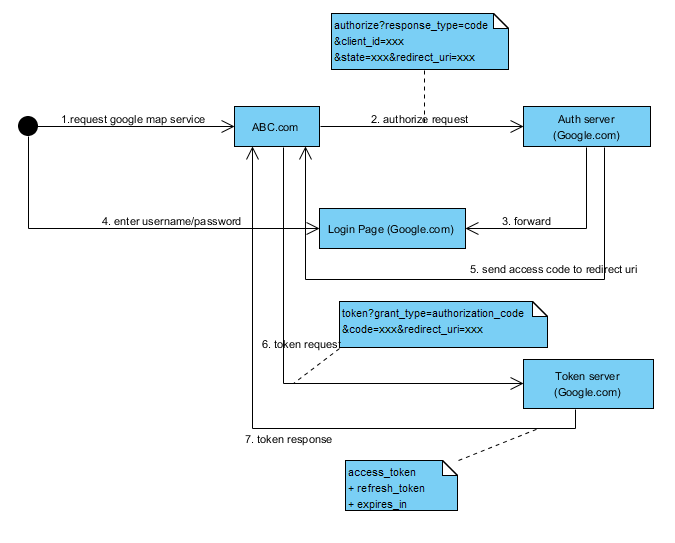
## Difference between client id/client secret and user name/password:

Say abc.com want to use google map service, it should register itself on google API platform and obtain a client id/client secret, so client id/client secret is **per site**. Another site that uses google map service has a different client id/client secret to identify itself.

User name/password, however, identifies a single user that wants to use google map service on abc.com. It should be a registered user on google.com

## Authorization code:

Authorization Flow:



Notes:

User on ABC.com can use google webservice without exposing its user name/password to ABC.com, reducing security risk

Use access code to exchange for an access token. The access code is effective only once. This prevents replay attack and reduce risk even if access code is exposed to attackers.

Requires user interaction: enter username/password and approve the authorization.

Configuration:

<oauth:authorization-code disabled=”true/false” authorization-code-services-ref=”xxx”/>

To access the Authorization Endpoint, user must be fully authenticated. Authentication is not the responsibility of OAuth. Therefore, we need an additional spring security filter chain in front of the authorization endpoint, which is responsible for: 1. displaying login page if user is not authenticated. 2. Authenticates user name and password

Eg.

<sec:http disable-url-rewriting="true" use-expressions="false">

<sec:csrf disabled="true"/>

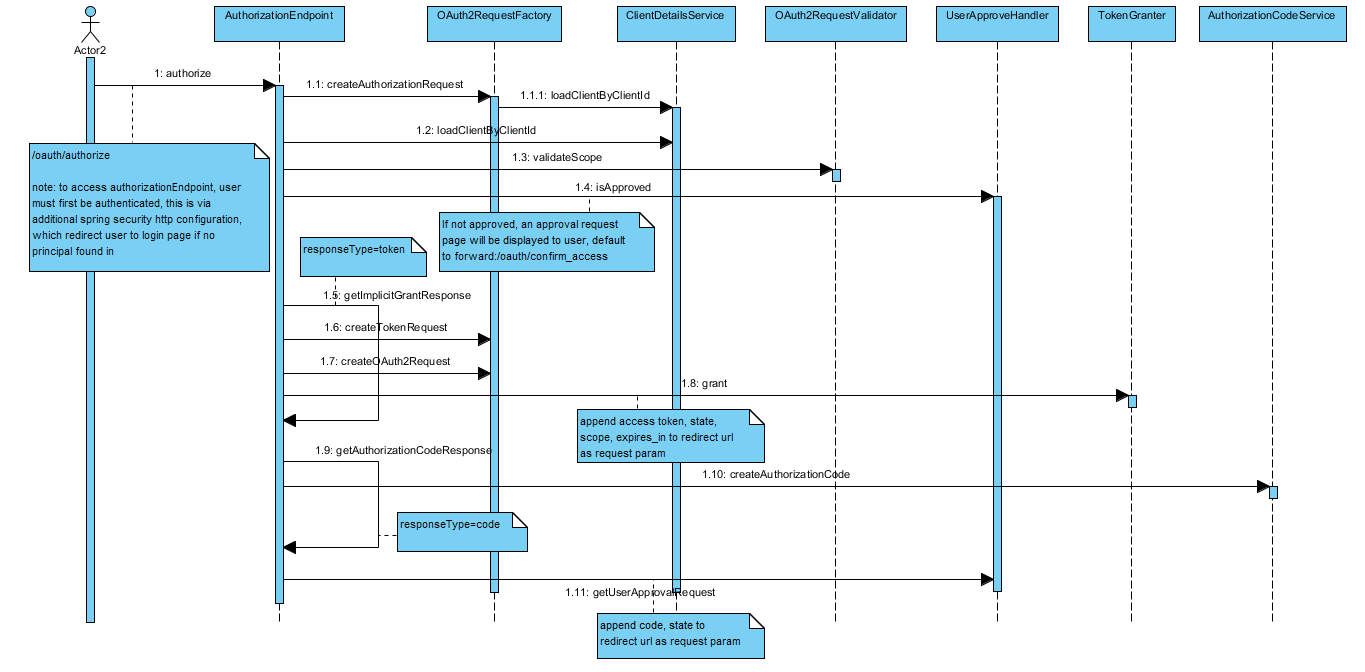
<sec:access-denied-handler error-page="/login.jsp?authorization\_error=true"/>

<sec:intercept-url pattern="/oauth/\*\*" access="IS\_AUTHENTICATED\_FULLY" requires-channel=”https”/>

<sec:form-login authentication-failure-url="/login.jsp?authentication\_error=true" default-target-url="/index.jsp" login-page="/login.jsp" login-processing-url="/login.do" />

</sec:http>

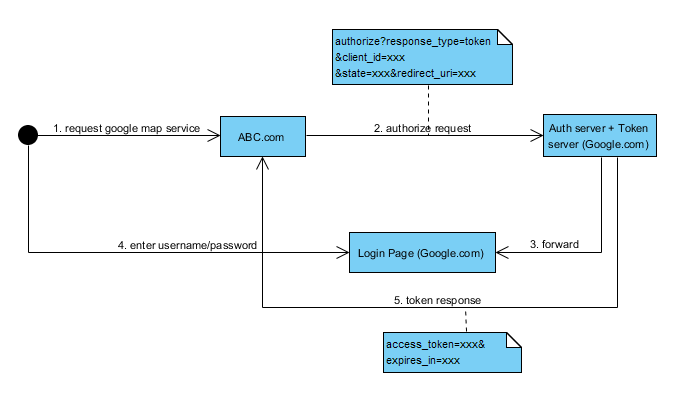
Spring security Sequence diagram:



The approve/deny decision is handled by AuthorizationEndpoint.approveOrDeny

## Implicit:

Authorization Flow:



Notes:

The authorization server issues authorization token directly without generating an access code first. The authorization server/token server responsibility is combined

Requires user interaction: enter username/password and approve the authorization.

Configuration:

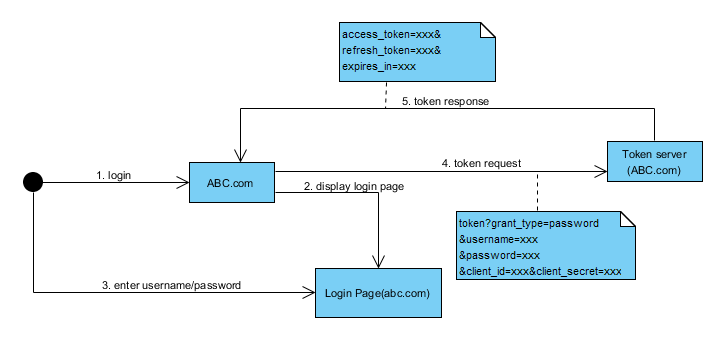
<oauth:implicit disabled=”true/false” />

Spring security Sequence diagram:

Same as above, different logic branch based on “response\_type” parameter of auth request.

## Password:

Authorization Flow:



Notes:

Request is sent to token server directly.

No user interaction is required for the authorization, so client actually knows resource server’s user name/password. Useful for building SOA application, say site A’s rest controller invoke some other site A’s restful service.

Configuration:

<oauth:password disabled=”true/false” authentication-manager-ref=”xxx” />

<security:http pattern="/oauth/token" disable-url-rewriting="true" use-expressions="false" create-session="stateless" authentication-manager-ref="oauth2ClientIdAuthenticationManager" entry-point-ref="oauth2AuthenticationEntryPoint">

<security:csrf disabled="true"/>

<security:intercept-url pattern="/oauth/token" access="IS\_AUTHENTICATED\_FULLY"/>

<security:anonymous enabled="false"/>

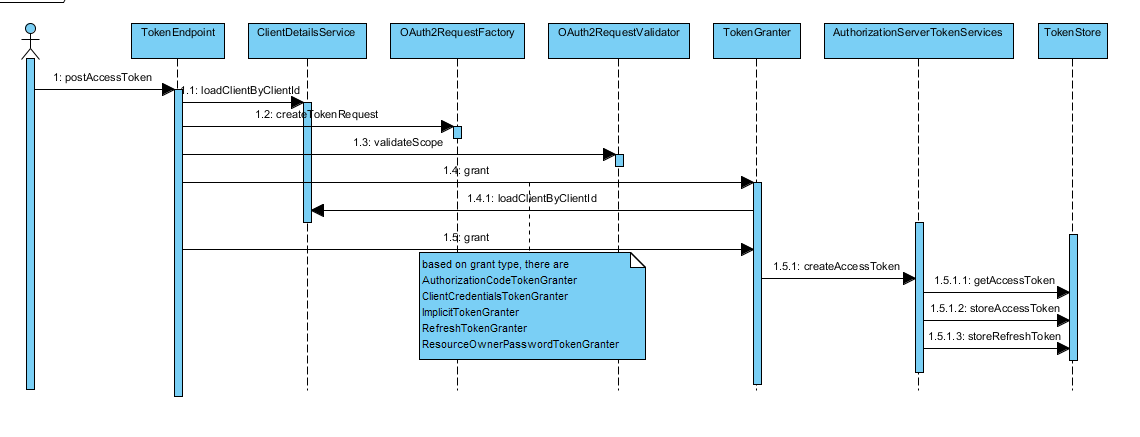
<security:http-basic entry-point-ref="oauth2AuthenticationEntryPoint"/>

<security:access-denied-handler ref="oauth2AccessDeniedHandler"/>

</security:http>

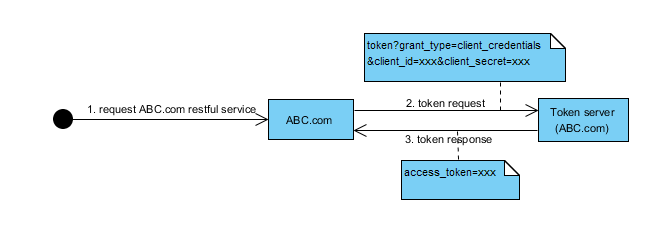
A spring security filter chain should be placed in front of the token endpoint. It is responsible for authenticating client id/secret with “oauth2ClientIdAuthenticationManager” which uses a ClientDetailService internally

Spring security Sequence diagram:



## Client credentials:

Authorization Flow:



Notes:

Request is sent to token server directly.

Does not even require user name/password, so every user will obtain the same auth token

Configuration:

<oauth:client-credentials disabled=”true/false” />

Spring security Sequence diagram:

Same as “password” authorization

## Authorization header verification:

Configuration:

<oauth:resource-server id="prodResourceAuthTokenChecker" resource-id="productdomain" token-services-ref="tokenServices" authentication-manager-ref="oAuth2AuthenticationManager" />

<security:http use-expressions="false" pattern="/\*\*" create-session="never" entry-point-ref="oauth2AuthenticationEntryPoint" access-decision-manager-ref="oauth2AccessDecisionManager"

authentication-manager-ref="oAuth2AuthenticationManager" >

<security:anonymous enabled="false"/>

<security:intercept-url pattern="/\*\*" access="IS\_AUTHENTICATED\_FULLY"/>

<security:custom-filter ref="prodResourceAuthTokenChecker" before="PRE\_AUTH\_FILTER"/>

<security:access-denied-handler ref="oauth2AccessDeniedHandler"/>

</security:http>

Notes:

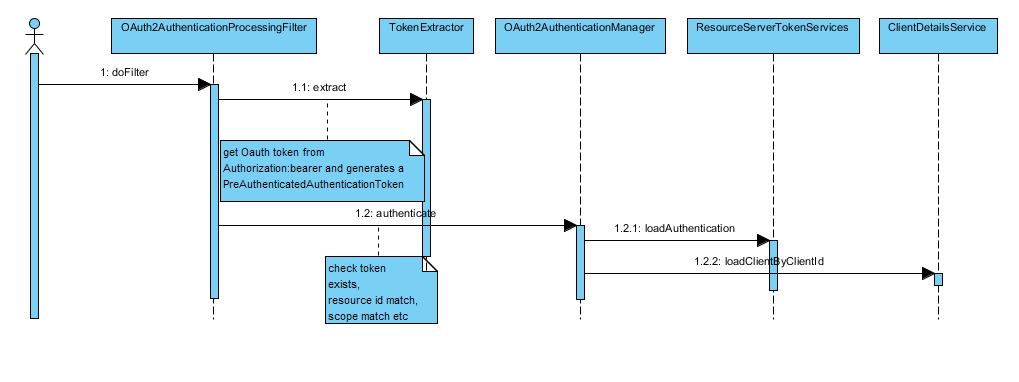
An “Auth:bearer” header representing the oauth token is carried when invoking the actual web service. A spring security filter needs to be placed in front of the web service controller to verify the oauth token and populates the authentication with user info

The <resource-server> declaration actually creates an instance of OAuth2AuthenticationProcessingFilter which uses a OAuth2AuthenticationManager internally. OAuth2AuthenticationManager is responsible for loading token and client details and verify them. An OAuth2Authentication will be populated in security context

The resource-server’s resource-id attribute should match the “resource\_ids” configuration of client detail.

oauth2AccessDecisionManager injects a org.springframework.security.oauth2.provider.vote.ScopeVoter which matches the configuration attributes with the “scope” configuration for client detail.

Spring security Sequence diagram:



## Overall Oauth architecture for SOA application:

