

- 2 types of tokens should be there: **access token** (short validity; around 10 min) and **refresh token** (long validity; like 6 months)
  - ❖ When the *access token* is expired, the **refresh token** will be used to get a new *access token*.
  - ❖ Usually, **refresh token** is stored in the cookies, and get the *refresh token* from the cookies from backend (not in request payload) and validate the *refresh token*;
  - ❖ If the *refresh token* is valid then generate one new **access token** and return in response.
  - ❖ Use **http-only** cookies to store refresh token.

```
LoginResponseDto responseDto = authService.login(loginDto);

Cookie cookie = new Cookie("refreshToken", responseDto.getRefreshToken());
cookie.setHttpOnly(true);
cookie.setSecure(true);

response.addCookie(cookie);

return ResponseEntity.ok(responseDto);
```

- ❖ Just keep **user id** in the refresh token, no need to keep everything there.

```
public String generateRefreshToken(@NotNull User user) { 1 usage @ Alok Ranjan Joshi
    return Jwts.builder()
        .subject(s: user.getId().toString())
        .issuedAt(date: new Date())
        .expiration(date: new Date(date: System.currentTimeMillis() + 1000L * 60 * 60 * 24 * 30 * 6))
        .signWith(getSecretKey())
        .compact();
}
```

- ❖ **Login service method** should get the user from **principal** not from *database*

```
Authentication authentication = authenticationManager.authenticate(
    authentication: new UsernamePasswordAuthenticationToken(principal: loginDto.getEmail(), credentials: loginDto.getPassword()));

User user = (User) authentication.getPrincipal();
String accessToken = jwtService.generateAccessToken(user);

String refreshToken = jwtService.generateRefreshToken(user);

return new LoginResponseDto(accessToken, refreshToken);
```

- ❖ For this, **user** should be stored as principal instead of **username**.

## ➤ Frontend & Backend in Google OAuth2 (in general)

- In case of “sign in with google”, There are **2 backends** involved
  - ~ Our own backend
  - ~ Google’s backend
- After clicking on that “sign in with google” button, the **frontend** triggers one request which is:
  - ~ GET /oauth2/authorization/google (this is our backend’s path; not google’s)
  - ~ We don’t write this end point by our self; this is **auto-created** by **Spring Security OAuth2 Client**.
  - ~ Internally it is handled by: **OAuth2AuthorizationRequestRedirectFilter**
- Now, our backend reads from **application.yml** file

```
spring:  
  security:  
    oauth2:  
      client:  
        registration:  
          google:  
            client-id: xxx  
            client-secret: yyy  
            scope: email, profile
```

- ~ Now, our backend knows about:
  - ~ which provider (google)
  - ~ client\_id
  - ~ scopes
  - ~ redirect\_uri template
- Now, our backend will build a **url having proper query parameters which will be sent as response to which the browser will redirect.**
  - ~ The URL will look something like this

```
https://accounts.google.com/o/oauth2/v2/auth  
?client_id=123456789.apps.googleusercontent.com  
&response_type=code  
&scope=openid%20email%20profile  
&redirect_uri=https://your-backend.com/login/oauth2/code/google  
&state=KJH7823HJDS
```

- ~ Note: **redirect\_uri** is present in this URL.

- Now, our Backend responds with one thing only

```
302 Redirect
```

```
↳ Location: https://accounts.google.com/o/oauth2/v2/auth?client_id=...
```

- ~ It is browser's rule: **If response status is 3xx and Location header is present → automatically navigate to that URL.**
- ~ There is no involvement of **frontend** in this case.
- Now, the browser redirects to **accounts.google.com** .... page directly.
  - ~ Here there is no role of our backend, it is completely backed by **Google's backend**.
- Now, the user will interact with the **google's UI** and give the necessary permissions of the required details.
- Then, Google will authenticate the user and redirect **the browser** to **our backend's path** Using the **same redirect\_uri** that your backend sent earlier

```
↳ https://your-backend.com/login/oauth2/code/google?code=XYZ&state=ABC
```

~ This full path (not just backend's domain) is registered in Google console.

~ Note: **state & code** is present here.

#### ↳ NOTE -----

- ~ So, in the Google Console you can register as many URIs you want, that will be treated as the allowed redirect URIs
- ~ And you need to pass the **redirect\_uri** in the response so that google will get to know to which **uri** it has to redirect.
- ~ First Google will check if the given **uri** in the request URL is present in the access list, if present then it'll redirect back to that URI after authenticating the user.

~

- Now, our backend's end point “/login/oauth2/code/google” will receive the request.

- ~ This endpoint is ALSO **auto-created** by **Spring Security OAuth2 Client**
- ~ Handled by: **OAuth2LoginAuthenticationFilter**

- Now, our backend validate the **state**

- ~ **state** is a random, unpredictable value generated by YOUR backend before redirecting the user to Google.

```
state = a8f9c2e4b3...
```

- ~ If the **state** matches, then continue; otherwise **reject**.

- Now **our Backend** exchanges **code** with **tokens**

- ~ **code** is an authorization code issued by Google after the user successfully authenticates and consents.

- ~ Think of it as a one-time, short-lived voucher that your backend can exchange for **token**.
- ~ Backend makes **server-to-server** call.
- ~ It sends **client\_id**, **client\_secret**, **code**, **redirect\_uri** and receives **access\_token**, **id\_token**, **expires\_in**

```
{
  "access_token": "...",
  "id_token": "...",
  "expires_in": 3600
}
```

- ~ **id\_token** is a JWT which contains the user details within it like the below

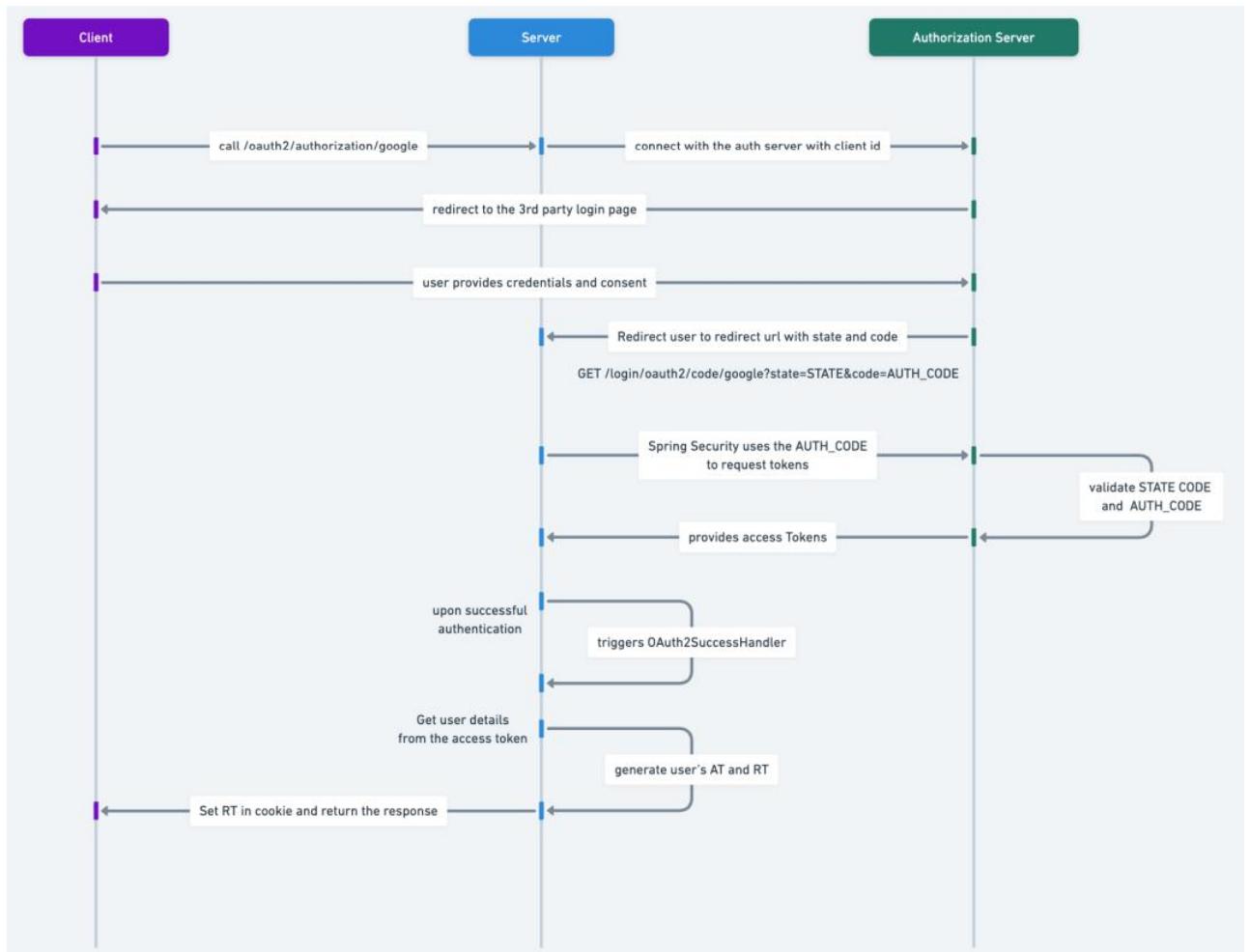
```
{
  "iss": "https://accounts.google.com",
  "sub": "109876543210",
  "email": "user@gmail.com",
  "email_verified": true,
  "name": "Alok Joshi",
  "picture": "...",
  "aud": "your-client-id",
  "exp": 1710000000
}
```

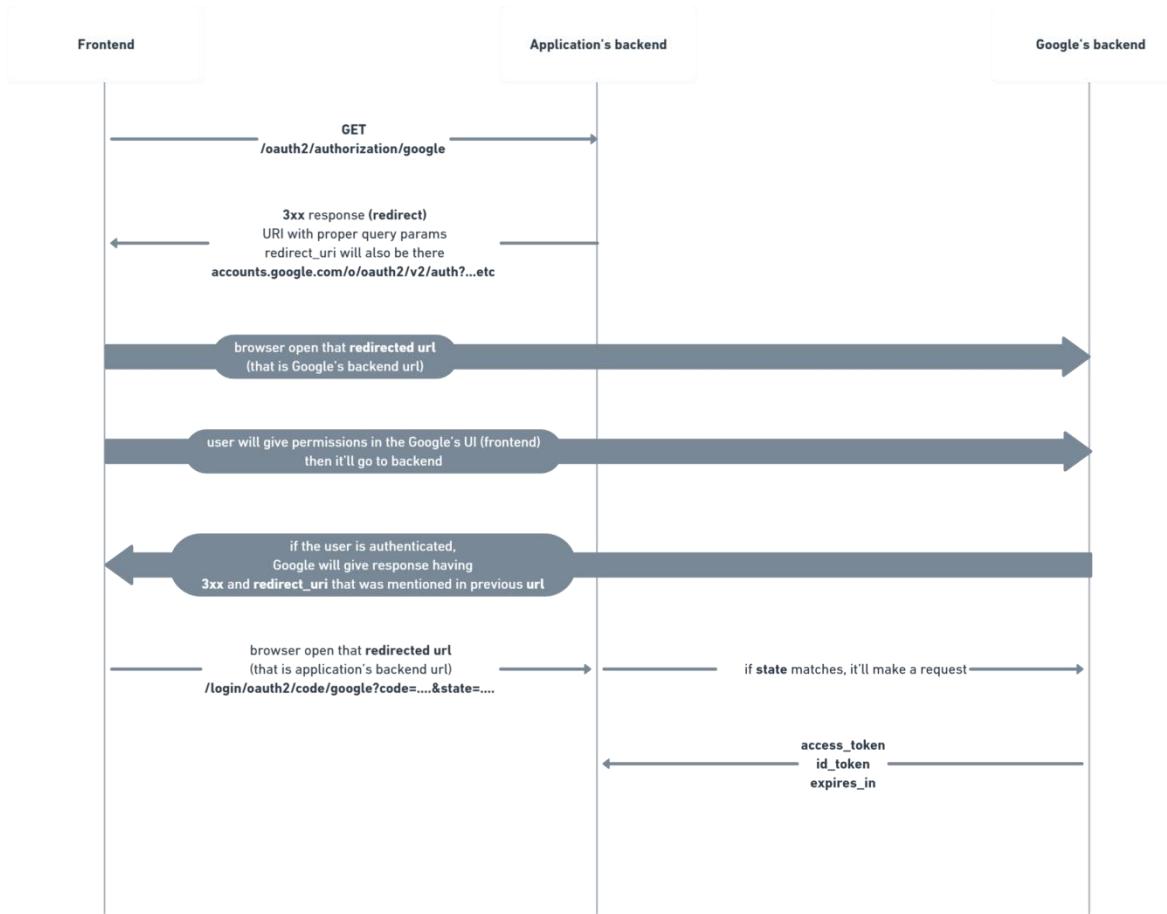
- ~ You “use” only the authorization code; the **access\_token** is returned because **OAuth** requires it, but if you don’t call Google APIs, it is perfectly normal that it is never used.
- Now, you can implement the normal login in our backend after getting those details.
- In one sentence:
  - ~ In Google OAuth login, the frontend only triggers navigation. The backend constructs the authorization request, redirects the browser to Google, and later receives the authorization code. Google authenticates the user, validates the redirect URI against its allow-list, and redirects back to the backend. The backend validates state, exchanges the code for tokens, extracts user identity, and then performs normal application login logic.

```

Frontend
↓
/oauth2/authorization/google (our backend)
↓
accounts.google.com (Google backend)
↓
/login/oauth2/code/google (our backend)
↓
frontend/dashboard

```





Made with  Whimsical

1 `id_token` contains the details that are required like email, name etc etc

1 if http status is `3xx` and `location` is present in response then browser will redirect that by default

Made with  Whimsical

## ➤ Steps

### ➤ Google Console Setup

- Select the project in Google Console; and go to Dashboard
- API & Services >> Credentials

#### API keys

<input type="checkbox"/>	<input checked="" type="radio"/>	Name	Bound account <small>?</small>	Creation date <small>↓</small>
No API keys to display				

#### OAuth 2.0 Client IDs

<input type="checkbox"/>	Name	Creation date <small>↓</small>
No OAuth clients to display		

#### Service Accounts

<input type="checkbox"/>	Email	Name <small>↑</small>
No service accounts to display		

### ➤ **Create Credentials >> OAuth Client ID**

#### ➤ **Configure Consent Screen**

- ↪ You need to configure all the things separately.
- ↪ Audience: either you can setup some test user or publish. Publish means anyone can try to login.
- ↪ Data access: it is the scopes like which data do you want to access from google.
- ↪ Clients: create the OAuth2 client;

#### Authorised JavaScript origins ?

For use with requests from a browser

URIs 1 \* \_\_\_\_\_  
http://localhost:8080

URIs 2 \* \_\_\_\_\_  
http://localhost

**+ Add URI**

\* It means Requests coming from a browser whose origin is

“http://localhost:8080” are allowed to start OAuth.

#### Authorised redirect URIs ?

For use with requests from a web server

URIs 1 \* \_\_\_\_\_  
http://localhost:8080/login/oauth2/code/google

**+ Add URI**

(default URL of spring security)

- After creating the **Client**, you'll get the **client id** and **client secret**, copy those and paste in the *application.properties* or *application.yml* file.

- ----- [Google Console set-up done](#) -----

- [Application codes](#)

- Add the **oauth2Login** filter in the custom filter chain

```
httpSecurity
    .authorizeHttpRequests( authorizeHttpRequestsCustomizer: auth -> auth
        .requestMatchers( ...patterns: "/auth/**").permitAll()
        .anyRequest().authenticated()
    .csrf( csrfCustomizer: csrfConfig -> csrfConfig.disable())
    .sessionManagement( sessionManagementCustomizer: sessionManagementConfig -> sessionManagementConfig
        .sessionCreationPolicy( sessionCreationPolicy: SessionCreationPolicy.STATELESS)
    .addFilterBefore(jwtFilter, beforeFilter: UsernamePasswordAuthenticationFilter.class)
    .oauth2Login( oauth2LoginCustomizer: oauth2Config -> oauth2Config
        .failureUrl( authenticationFailureUrl: "/login?error=true"));
    .
```

- You need to also add the **success** handler, otherwise even after getting the response from authorization server (google in our case) it'll not do the required things after authenticating the user.
  - ~ There is a class **SimpleUrlAuthenticationSuccessHandler**, that decides what to do after the authentication success.
  - ~ Authentication can be of any type i.e. **OAuth**, **Form login**, **Username/password login** ..etc
  - ~ This class contains one method **onAuthenticationSuccess** which is executed after the authentication gets succeeded.
  - ~ So, if we create a **Bean** of a class extending **SimpleUrlAuthenticationSuccessHandler** class and overriding the method **onAuthenticationSuccess** then we can handle the authentication success case.
  - ~ I created the below class:

```
@Slf4j 2 usages
@Component
@RequiredArgsConstructor
public class OAuth2SuccessHandler extends SimpleUrlAuthenticationSuccessHandler {
```

- Now just add that class's object in the SecurityFilterChain

```
.oauth2Login( oauth2LoginCustomizer: oauth2Config -> oauth2Config
    .failureUrl( authenticationFailureUrl: "/login?error=true")
    .successHandler(oAuth2SuccessHandler));
```

- I just checked if the user is present in the database, if present then do signup otherwise just create **access** and **refresh** token and send in response.

```

if(user == null) {
    User newUser = User.builder()
        .email(email)
        .name(oAuth2User.getAttribute("name"))
        .build();
    user = userService.save(newUser);
}

String accessToken = jwtService.generateAccessToken(user);
String refreshToken = jwtService.generateRefreshToken(user);

Cookie cookie = new Cookie("refreshToken", refreshToken);
cookie.setHttpOnly(true);
cookie.setSecure("production".equals(deployEnv));
response.addCookie(cookie);

String frontendUrl = "http://localhost:8080/home.html?token=" + accessToken;
response.sendRedirect(s: frontendUrl);

```

- It is inside the **onAuthenticationSuccess** method.
- **response.sendRedirect()** will send a redirect-response so that the browser will redirect to this specific url.
- **home.html** is nothing but a static file.
- F

## ➤ OAuth2 flow in Spring Security

- There are 2 filters which are important:
  - ~ **OAuth2AuthorizationRequestRedirectFilter**
    - backend to google before authentication
  - ~ **OAuth2LoginAuthenticationFilter**
    - google to backend after authentication
- **OAuth2AuthorizationRequestRedirectFilter** consists of 2 things
  - ~ One is to create the final object which is having **state, client id, client secret, redirect uri** etc etc every details that should be present in the request (for backend to google redirect).
    - This final object is of type **OAuth2AuthorizationRequest**
  - ~ Another is to save that object while the control goes to google.
  - ~ It generates the default oauth login end point, like **/oauth2/authorization/google**

```
public class OAuth2AuthorizationRequestRedirectFilter extends OncePerRequestFilter { 21 usages
    public static final String DEFAULT_AUTHORIZATION_REQUEST_BASE_URI = "/oauth2/authorization";
    private final ThrowawayAnalyzer throwawayAnalyzer;
    private RedirectStrategy authorizationRedirectStrategy;
    private OAuth2AuthorizationRequestResolver authorizationRequestResolver;
    private AuthorizationRequestRepository<OAuth2AuthorizationRequest> authorizationRequestRepository;
    private RequestCache requestCache;
```

- There is a class called **ClientRegistration**,
  - ~ It contains everything like **clientId, clientSecret, registrationId** (google, github ..etc), **redirectUri**
  - ~ It doesn't contain state because it is generated after starting the application after reading the **application.yml** or **application.properties** file.
  - ~ Here **redirectUri** will be in format
    - "{baseUrl}/login/oauth2/code/{registrationId}"

```
public final class ClientRegistration implements Serializable {
    private static final long serialVersionUID = 620L;
    private String registrationId;
    private String clientId;
    private String clientSecret;
    private ClientAuthenticationMethod clientAuthenticationMethod;
    private AuthorizationGrantType authorizationGrantType;
    private String redirectUri;
    private Set<String> scopes = Collections.emptySet();
    private ProviderDetails providerDetails = new ProviderDetails();
    private String clientName;
```

- The objects of **ClientRegistration** are stored in **ClientRegistrationRepository**.
  - ~ **InMemoryClientRegistrationRepository** implements **ClientRegistrationRepository** is having one *Map* of *registrationId* to *ClientRegistration*
    - ~ github : clientRegistration
    - ~ google : clientRegistration
    - ~ .... like this

```
public final class InMemoryClientRegistrationRepository implements ClientRegistrationRepository {
    private final Map<String, ClientRegistration> registrations;
```

- Now it comes **OAuth2AuthorizationRequestResolver**
  - ~ It creates objects of type **OAuth2AuthorizationRequest** using **ClientRegistration** object.
  - ~ It contains **state** as well and remaining things that ClientRegistration contains.
  - ~ Here the *redirectUri* will be in the form:
    - ~ "http://localhost:8080/login/oauth2/code/google"
    - ~ It is complete URL.
- Now, the final object is there, it just needs to be stored in a safe place for future use. **AuthorizationRequestRepository** is used to store that.
  - ~ It doesn't store in any *map* or *list*, it just stores the AuthorizationRequest object in the **session** of the **request** object.

```
public void saveAuthorizationRequest( @Nullable OAuth2AuthorizationRequest authorizationRequest, @Nullable OAuth2AuthorizationResponse response) {
    Assert.notNull( object: request, message: "request cannot be null");
    Assert.notNull( object: response, message: "response cannot be null");
    if (authorizationRequest == null) {
        this.removeAuthorizationRequest(request, response);
    } else {
        String state = authorizationRequest.getState();
        Assert.hasText( text: state, message: "authorizationRequest.state cannot be empty");
        request.getSession().setAttribute( s: this.sessionAttributeName, o: authorizationRequest);
    }
}
```

- Now, everything is done; our backend will send a redirect response to frontend, after seeing which the browser will redirect to that particular url (accounts.google.com ....)
- **OAuth2LoginAuthenticationFilter**
  - ~ It contains an object of type **AuthorizationRequestRepository** and gets the **OAuth2AuthorizationRequest** object from it.
  - ~ It gets the response from OAuth2 provider and validate that with **OAuth2AuthorizationRequest** object, like **state** matching and all.
  - ~ If any mismatch is there then it'll reject that.

- ~ Then it creates one token of type **OAuth2LoginAuthenticationToken** (just like **UsernamePasswordAuthenticationToken** in **DaoAuthenticationProvider**) and call the **authenticationManager.authenticate()** method.
- ~ Now just like username password authentication flow, this **authenticate()** method call delegates to **AuthenticationProvider's authenticate()** method and here the **AuthenticationProvider** is **OAuth2LoginAuthenticationProvider** .
- ~ **OAuth2LoginAuthenticationProvider** is the one who exchanges the **code** to get the **tokens** .
  - ~ It is done with the help of **DefaultAuthorizationCodeTokenResponseClient**



- ~ Now, where is the filter that creates the default login page for username password login form, oauth2 default login page.
  - ~ The filter is **DefaultLoginPageGeneratingFilter**
  - ~ It generates the default login pages.

```
private @NotNull String generateLoginPageHtml(@NotNull HttpServletRequest request, boolean loginError, boolean log
    String errorMsg = loginError ? this.getLoginErrorMessage(request) : "Invalid credentials";
    String contextPath = request.getContextPath();
    StringBuilder sb = new StringBuilder();
    sb.append("<!DOCTYPE html>\n");
    sb.append("<html lang=\"en\"\n");
    sb.append("  <head>\n");
    sb.append("    <meta charset=\"utf-8\"\n");
    sb.append("    <meta name=\"viewport\" content=\"width=device-width, initial-scale=1, shrink-to-fit=no\"\n");
    sb.append("    <meta name=\"description\" content=\"\"\n");
    sb.append("    <meta name=\"author\" content=\"\"\n");
    sb.append("    <title>Please sign in</title>\n");
    sb.append("    <link href=\"https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css\" rel=\n");
    sb.append("    <link href=\"https://getbootstrap.com/docs/4.0/examples/signin/signin.css\" rel=\"stylesheet\"\n");
    sb.append("  </head>\n");
    sb.append("  <body>\n");
    sb.append("    <div class=\"container\"\n");
    if (this.formLoginEnabled) {...}

    if (this.oauth2LoginEnabled) {...}

    if (this.saml2LoginEnabled) {...}
  
```

- ~ F



- ~ The redirected URIs are matches at the **filter level**, not **servlet level**. Spring Security is completely *filter based*.

```
if (request.getRequestURI().equals("/oauth2/authorization/google")) {
    // handle here
}
```

- ~

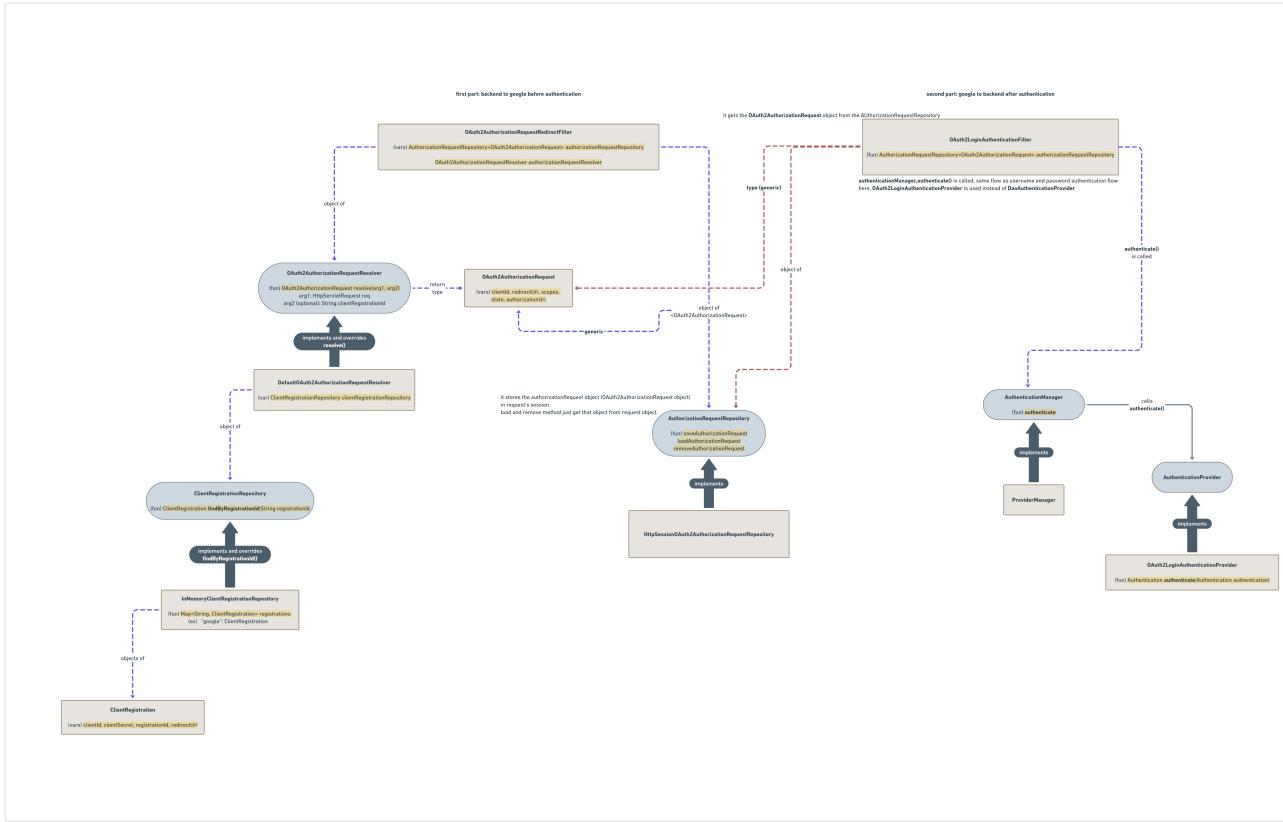
➤

### OAuth2 work-flow in general (I am taking google as OAuth2 provider)

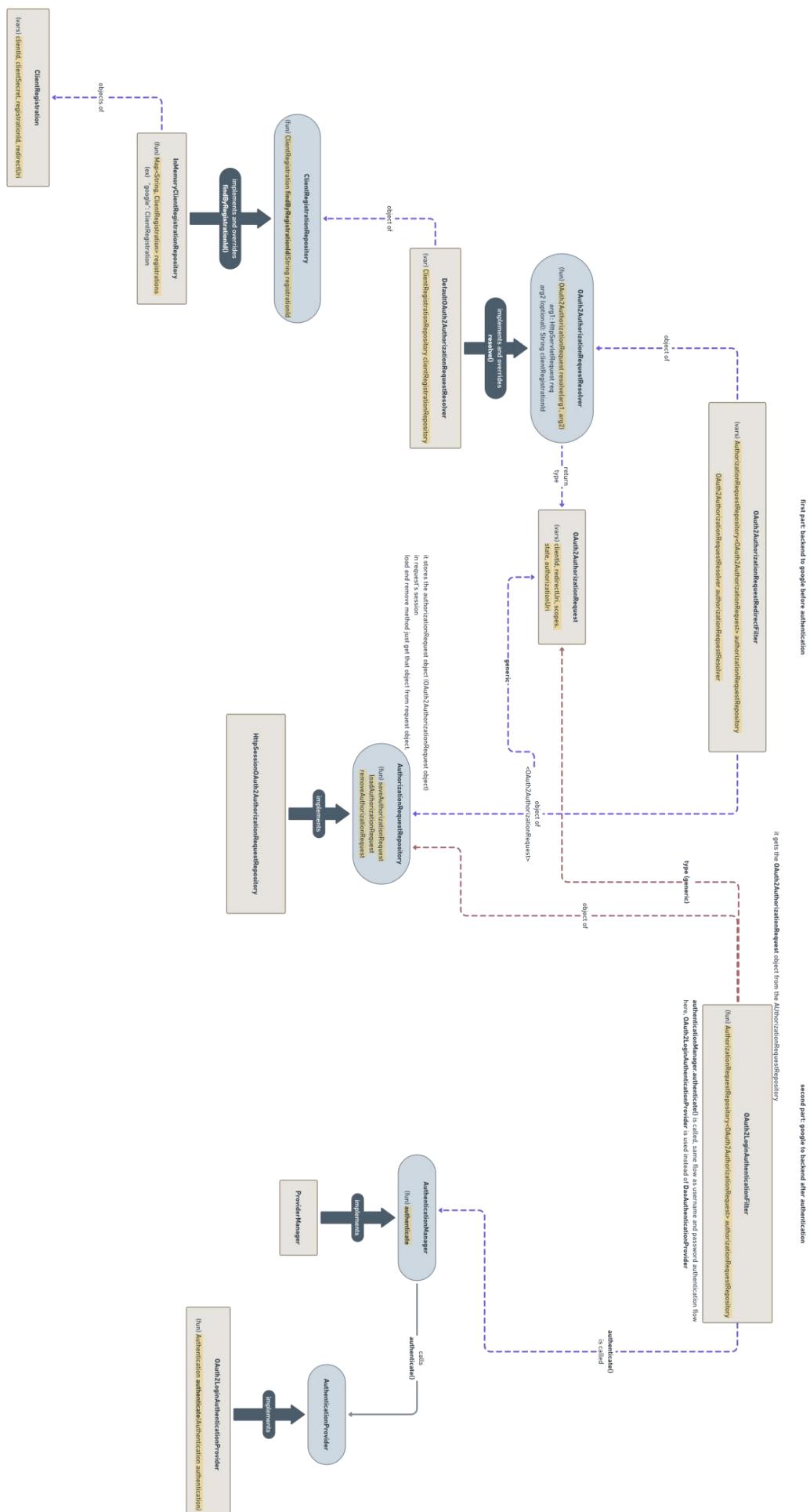
- 2 backend are involved here: **google's backend, our application backend**
- first, when the **signin with google** (or any other provider) button is clicked in the frontend, it'll call a **GET request to Application Backend**
  - in spring, this url is: **oauth2/authorization/google**
- now, **Application Backend** creates a proper url having client\_id, client\_secret, redirect\_uri, state etc etc.
  - note **redirect\_uri** and **state** here, it'll be used later....

```
https://accounts.google.com/o/oauth2/v2/auth
?client_id=123456789.apps.googleusercontent.com
&response_type=code
&scope=openid%20email%20profile
&redirect_uri=https://your-backend.com/login/oauth2/code/google
&state=KJH7823HJD5
```

status code **3xx** (which is meant for redirect)
- every browser's default behaviour is that if the response contains http status code **3xx** and there is a **location** field there, then it'll redirect it to that location; there is no involvement of frontend here, its completely at the browser level.
- now the browser will redirect to **accounts.google.com/.....** and user will get a screen to select the google account, and grant the consents.
- After authenticating successfully, **google** will send a response (**3xx** and **location**) and location will contain that **redirect\_uri** which was sent earlier.
- so now, browser will redirect to that **redirect\_uri** which goes to **Application Backend**.
  - this **redirect\_uri** will be
    - "https://www.your-backend.com/login/oauth2/code/google?code=abc&state=xyz"
- now this **state** will be checked, it should match with the **state** that was sent to google before authentication.
- if validation are successful, then **application backend** will use that **code** and make one **POST** request to google to get the **tokens**.



Made with Whimsical



Made with  Whimsical

### Part-1: application backend to oauth2 provider's backend

- First the request come to **OAuth2AuthorizationRequestRedirectFilter**
- It contains 2 parts:
  - one part builds the **OAuth2AuthorizationRequest** object (which is the final object which will be sent to oauth2 provider)
  - another part will save that object when the control goes to OAuth2 provider.
- **ClientRegistration** is a class that contains all the details of the Client i.e. client id, client secret, registrationId, redirectUri etc; but it *doesn't contain the state*.
  - here **redirectUri** is in the form: "{baseUrl}/login/oauth2/code/{registrationId}"
- There is a repository that stores all the **ClientRegistration** objects, which is **ClientRegistrationRepository**.
  - It contains a method **findById** which gives the ClientRegistration object for the particular registrationId (registrationId is nothing but "google", "facebook", "github")
- So, now we have all the objects os **ClientRegistration**, but we need to make the object proper to send in the request to OAuth2 provider.
  - because it has only templates kind of things. like you can see **redirectUri** is just a template, there is no **state** present here.
  - So, there is a **resolver** class which is named as **DefaultOAuth2AuthorizationRequestResolver** which takes the **ClientRegistration** object and create a **OAuth2AuthorizationRequest** object which contains fully ready values that can be sent to OAuth2 provider directly.
  - here **state** will be present.
  - here the **redirectUri** will be in the form of: "<http://localhost:8080/login/oauth2/code/google>"
- Now, everything is completed, just the request has to be sent to OAuth2 provider; but we need to keep the object somewhere to validate after getting the response from OAuth2 provider.
  - So, there is a repository **AuthorizationRequestRepository** is there to store the **OAuth2AuthorizationRequest** object.
  - it doesn't store those objects in any list or map, it just creates one attribute in the **request** object and store there.

### Part-2: oauth2 provider to application's backend

- **OAuth2LoginAuthenticationFilter** is the first filter in this case.
- it gets the **OAuth2AuthorizationObject** from the **AuthorizationRequestRepository** .
- then it validate the details from the response of OAuth2 provider with the **OAuth2AuthorizationObject**, for example **state** will be checked.
- if any mismatch is there, then it'll reject it.
- after that it'll simply call the **authenticate()** method of **authenticationManager**, just like we do in case of **username and password authentication**.
  - here the authentication token will be of type: **OAuth2LoginAuthenticationToken**
- then this **authentication()** method call will be delegated to **AuthenticationProvider's authentication()** method and in case of OAuth2, the **AuthenticationProvider** is **OAuth2LoginAuthenticationProvider**.
- this **OAuth2LoginAuthenticationProvider** exchange the **code** that was gotten from OAuth2 provider's response and get the **tokens** like **id\_token**, **accessToken** etc etc.
- this **id\_token** contains the necessary details about the user that we want like **username**, **email** etc etc.
- after that it just store the authentication object in security context and we can get the details using **getPrincipal()** method just like **username and password authentication**.

➤ F

➤ F

➤ F

➤ F

➤ F

➤