➤ 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.

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

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

response.addCookie(cookie);

return ResponseEntity.ok( body: responseDto);
```

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

```java
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*

```java
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**

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

- **id_token** is a JWT which contains the user details within it like the below
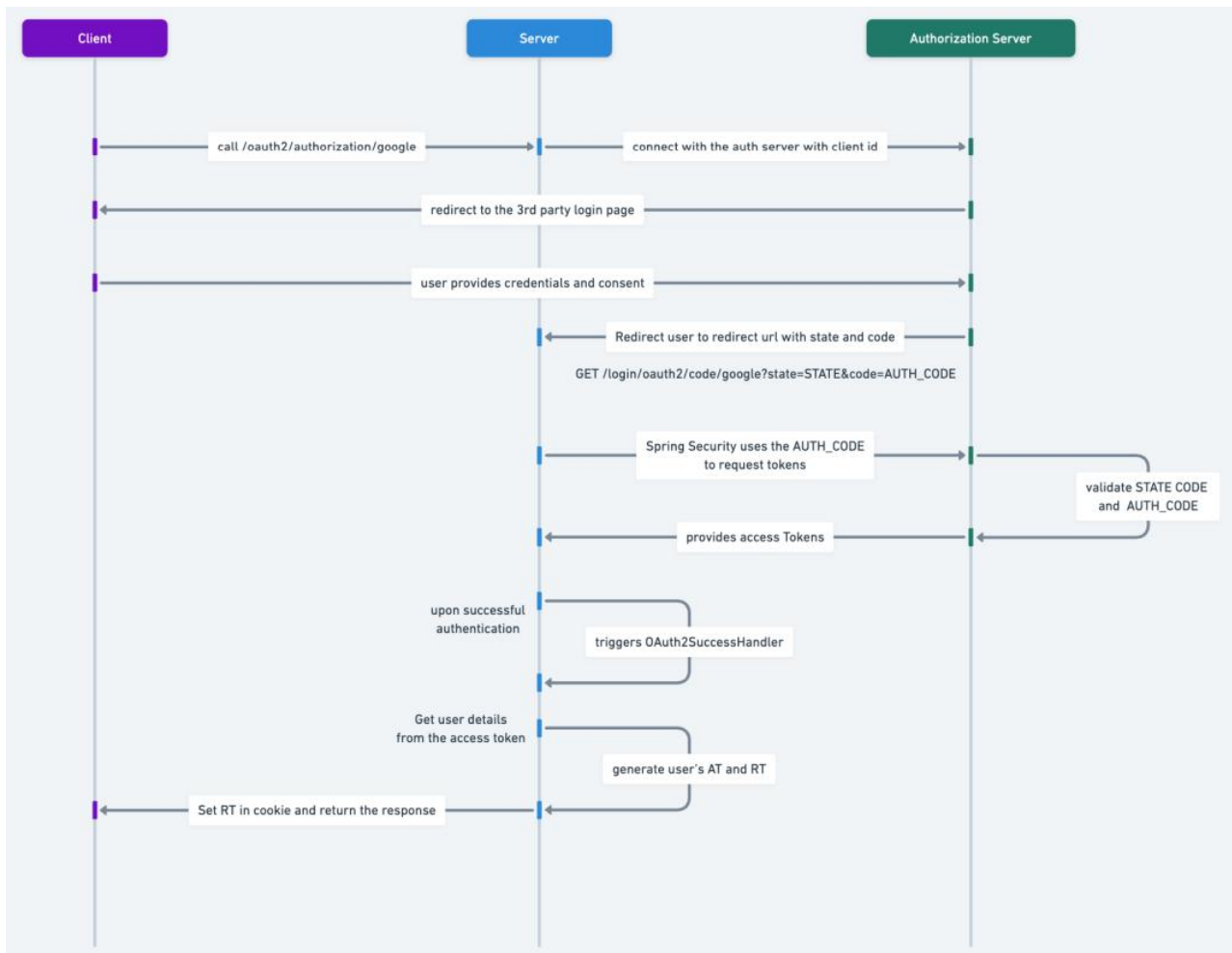
```json
{
    "iss": "https://accounts.google.com",
    "sub": "109876543210",
    "email": "user@gmail.com",
    "email_verified": true,
    "name": "Alok Joshi",
    "picture": "...",
    "aud": "your-client-id",
    "exp": 1710000000
}
```

- 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
```

➢

➢

Frontend	Application's backend	Google's backend

GET
/oauth2/authorization/google

3xx response **(redirect)**
URI with proper query params
redirect_uri will also be there
accounts.google.com/o/oauth2/v2/auth?...etc

browser open that **redirected url**
(that is Google's backend url)

user will give permissions in the Google's UI (frontend)
then it'll go to backend

if the user is authenticated,
Google will give response having
3xx and **redirect_uri** that was mentioned in previous **url**

browser open that **redirected url**
(that is application's backend url)
/login/oauth2/code/google?code=....&state=....

if **state** matches, it'll make a request

access_token
id_token
expires_in

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

		Name	Bound account ⑦	Creation date ↓
☐	●			

No API keys to display

OAuth 2.0 Client IDs

	Name	Creation date ↓
☐		

No OAuth clients to display

Service Accounts

	Email	Name ↑
☐		

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 ⑦

For use with requests from a browser

URIs 1 *
http://localhost:8080

URIs 2 *
http://localhost

    ع   + Add URI

           It means Requests coming from a browser whose origin is "http://localhost:8080" are allowed to start OAuth.

Authorised redirect URIs ⑦

For use with requests from a web server

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

    + Add URI

    ع   (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( name: oAuth2User.getAttribute( name: "name"))
            .build();
    user = userService.save(newUser);
}
String accessToken  = jwtService.generateAccessToken(user);
String refreshToken  = jwtService.generateRefreshToken(user);


Cookie cookie = new Cookie( name: "refreshToken",  value: 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

➢ <u>OAuth2AuthorizationRequestRedirectFilter</u> 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**

```java
public class OAuth2AuthorizationRequestRedirectFilter extends OncePerRequestFilter {  21 usages
    public static final String DEFAULT_AUTHORIZATION_REQUEST_BASE_URI = "/oauth2/authorization";
    private final ThrowableAnalyzer throwableAnalyzer;
    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 contains 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}"

```java
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 Cl
        private final Map<String, ClientRegistration> registrations;
    ```
  - ⌐
- ➢ Now it comes **OAuth2AuthorizationRequestResolver**
  - ⌐ It create 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 store the AuthorizationRequest object in the *session* of the *request* object.
    ```
    public void saveAuthorizationRequest( @Nullable OAuth2AuthorizationRequest authorizationRequest,  @N
        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 get the **OAuth2AuthorizationRequest** object from it.
  - ⌐ Then it create 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**

➢ --

➢ Now, where is the filter that create the default login page for username password login form, oauth2 default login page.

- The filter is **DefaultLoginPageGeneratingFilter**

- It generates the default login pages.

```java
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=
    sb.append("    <link href=\"https://getbootstrap.com/docs/4.0/examples/signin/signin.css\" rel=\"stylesheet\"
    sb.append("  </head>\n");
    sb.append("  <body>\n");
    sb.append("     <div class=\"container\">\n");
    if (this.formLoginEnabled) {...}

    if (this.oauth2LoginEnabled) {...}

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

-

- F

➢ **NOTE** ---

- The redirected URIs are matches at the **filter** level, not **servlet level**. Spring Security is completely *filter based*.
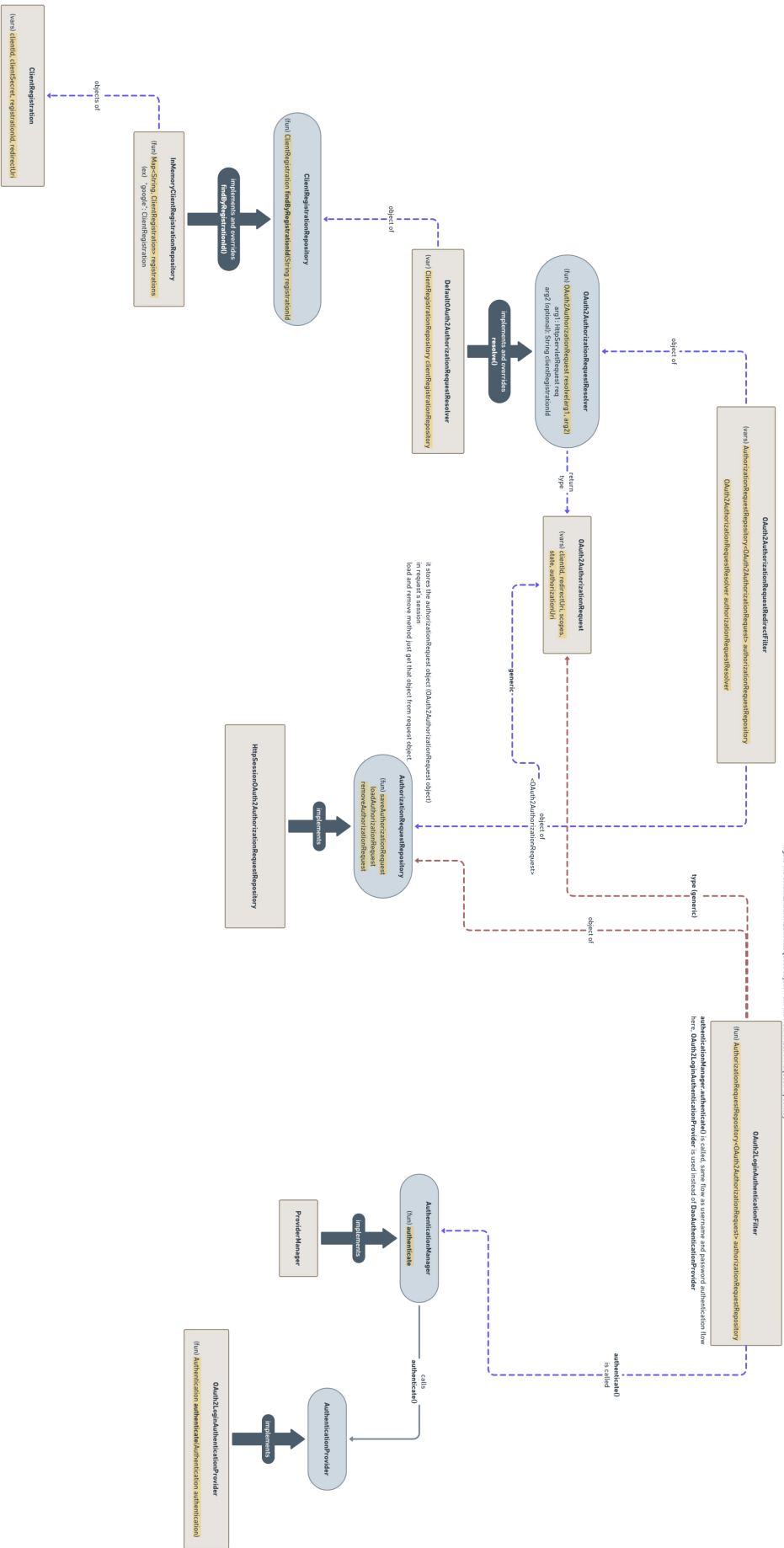
```java
if (request.getRequestURI().equals("/oauth2/authorization/google")) {
    // handle here
}
```

-

-

➢

➢ F

➢ F

➢ F

➢ F

ClientRegistration
(vars) clientId, clientSecret, registrationId, redirectUri

objects of

InMemoryClientRegistrationRepository
(fun) Map<String, ClientRegistrations
(ex) "google": ClientRegistration registrations

implements and overrides
findByRegistrationId()

ClientRegistrationRepository
(fun) ClientRegistration **findByRegistrationId**(String registrationId)

object of

DefaultOAuth2AuthorizationRequestResolver
(var) ClientRegistrationRepository clientRegistrationRepository

implements and overrides
resolve()

OAuth2AuthorizationRequestResolver
(fun) OAuth2AuthorizationRequest resolve(arg1, arg2)
arg1: HttpServletRequest req
arg2 (optional): String clientRegistrationId

object of

first part: backend to google before authentication

OAuth2AuthorizationRequestRedirectFilter
(vars) AuthorizationRequestRepository<OAuth2AuthorizationRequest>
OAuth2AuthorizationRequestRepository
OAuth2AuthorizationRequestResolver authorizationRequestResolver

return type

OAuth2AuthorizationRequest
(vars) clientId, redirectUri, scopes, state, authorizationUri

generic

it stores the authorizationRequest object (OAuth2AuthorizationRequest object) in request's session
load and remove method just get that object from request object

<OAuth2AuthorizationRequest>

object of

AuthorizationRequestRepository
(fun) saveAuthorizationRequest
loadAuthorizationRequest
removeAuthorizationRequest

implements

HttpSessionOAuth2AuthorizationRequestRepository

type (generic)

it gets the **OAuth2AuthorizationRequest** object from the AuthorizationRequestRepository

object of

second part: google to backend after authentication

OAuth2LoginAuthenticationFilter
(fun) AuthorizationRequestRepository<OAuth2AuthorizationRequest> authorizationRequestRepository

authenticationManager.authenticate() is called, same flow as username and password authentication flow
here, **OAuth2LoginAuthenticationProvider** is used instead of **DaoAuthenticationProvider**

authenticate()
is called

AuthenticationManager
(fun) **authenticate**

implements

ProviderManager

calls
authenticate()

AuthenticationProvider

implements

OAuth2LoginAuthenticationProvider
(fun) Authentication **authenticate**(Authentication authentication)

- F
- F
- F
- F
- F
-