OAuth Google login flow:

* The OAuth2 login flow will be initiated by the frontend client by sending the user to the endpoint [http://localhost:8080/oauth2/authorize/{provider}?redirect\_uri=](http://localhost:8080/oauth2/authorize/%7Bprovider%7D?redirect_uri=)<redirect\_uri\_after\_login>.

The provider path parameter is one of google, facebook, or github. The redirect\_uri is the URI to which the user will be redirected once the authentication with the OAuth2 provider is successful. This is different from the OAuth2 redirectUri.

* On receiving the authorization request, Spring Security’s OAuth2 client will redirect the user to the AuthorizationUrl of the supplied provider.

All the state related to the authorization request is saved using the authorizationRequestRepository specified in the SecurityConfig.

The user now allows/denies permission to your app on the provider’s page. If the user allows permission to the app, the provider will redirect the user to the callback url [http://localhost:8080/oauth2/callback/{provider](http://localhost:8080/oauth2/callback/%7Bprovider)} with an authorization code. If the user denies the permission, he will be redirected to the same callbackUrl but with an error.

* If the OAuth2 callback results in an error, Spring security will invoke the oAuth2AuthenticationFailureHandler specified in the above SecurityConfig.
* If the OAuth2 callback is successful and it contains the authorization code, Spring Security will exchange the authorization\_code for an access\_token and invoke the customOAuth2UserService specified in the above SecurityConfig.
* The customOAuth2UserService retrieves the details of the authenticated user and creates a new entry in the database or updates the existing entry with the same email.
* Finally, the oAuth2AuthenticationSuccessHandler is invoked. It creates a JWT authentication token for the user and sends the user to the redirect\_uri along with the JWT token in a query string.

**The classes added:**

1. **SecurityConfig** class: It contains configurations for both OAuth2 social login
2. **HttpCookieOAuth2AuthorizationRequestRepository:** The OAuth2 protocol recommends using a state parameter to prevent CSRF attacks. During authentication, the application sends this parameter in the authorization request, and the OAuth2 provider returns this parameter unchanged in the OAuth2 callback
3. **CustomOAuth2UserService**: extends Spring Security’s DefaultOAuth2UserService and implements its loadUser() method. This method is called after an access token is obtained from the OAuth2 provider.
4. **OAuth2UserInfo**: abstract class  get the required details of the user from the generic map of key-value pairs
5. **GoogleOAuth2UserInfo :** get the user info from google
6. **OAuth2UserInfoFactory** :
7. **OAuth2AuthenticationSuccessHandler** On successful authentication, Spring security invokes the onAuthenticationSuccess() method of the OAuth2AuthenticationSuccessHandler configured in SecurityConfig. this method, we perform some validations, create a JWT authentication token, and redirect the user to the redirect\_uri specified by the client with the JWT token added in the query string
8. **OAuth2AuthenticationFailureHandler**: In case of any error during OAuth2 authentication, Spring Security invokes the onAuthenticationFailure() method of the OAuth2AuthenticationFailureHandler that we have configured in SecurityConfig.
9. **AuthController**
10. **CustomUserDetailsService**
11. **TokenProvider** :  contains code to generate and verify Json Web Tokens
12. **TokenAuthenticationFilter**: is used to read JWT authentication token from the request, verify it, and set Spring Security’s SecurityContext if the token is valid
13. **RestAuthenticationEntryPoint**: class is invoked when a user tries to access a protected resource without authentication. In this case, we simply return a 401 Unauthorized response
14. **UserPrincipal** class represents an authenticated Spring Security principal. It contains the details of the authenticated user
15. **CurrentUser**: a meta-annotation that can be used to inject the currently authenticated user principal in the controllers
16. **UserController** class contains a protected API to get the details of the currently authenticated user
17. **CookieUtils**
18. **LoginRequest**:  request/response payloads are used in our controller APIs
19. **AuthResponse**
20. **ApiResponse**
21. **BadRequestException**
22. **ResourceNotFoundException**
23. **OAuth2AuthenticationProcessingException**
24. **WebMvcConfig:** enable CORS so that our frontend client can access the APIs from a different origin. I’ve enabled the origin [http://localhost:3000](http://localhost:3000/) since that is where our frontend application will be running.