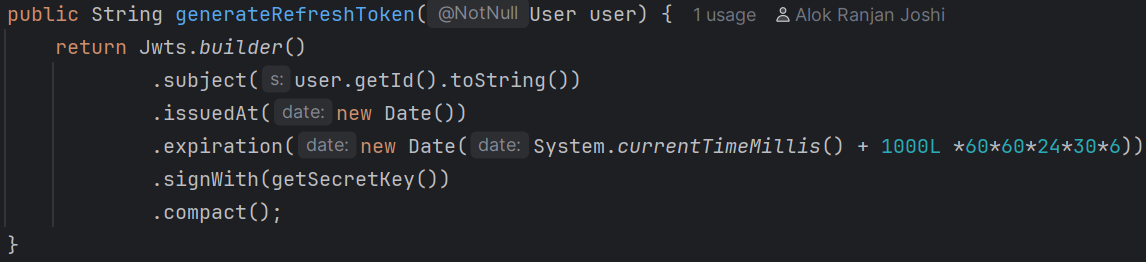
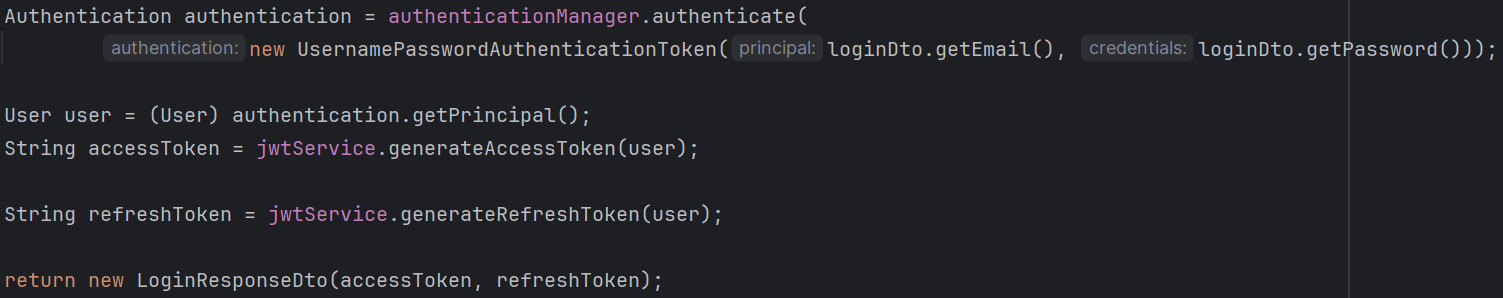
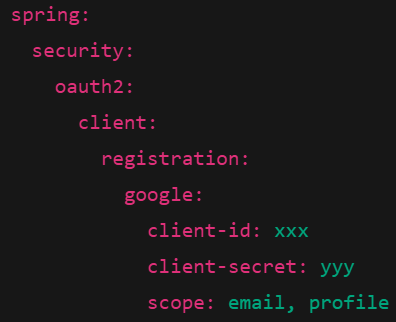
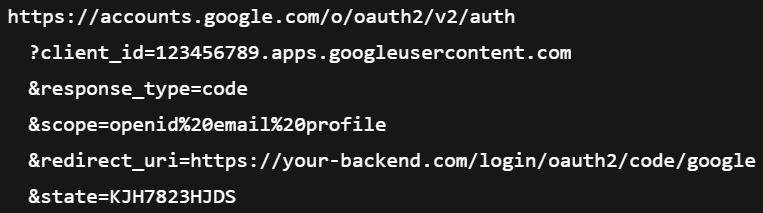
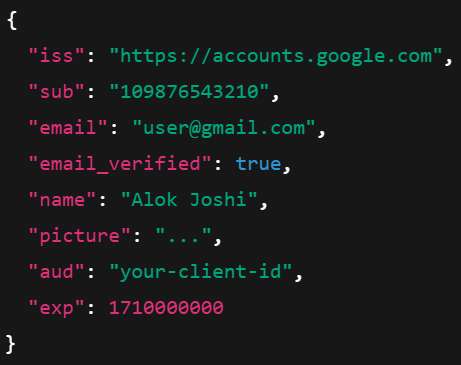
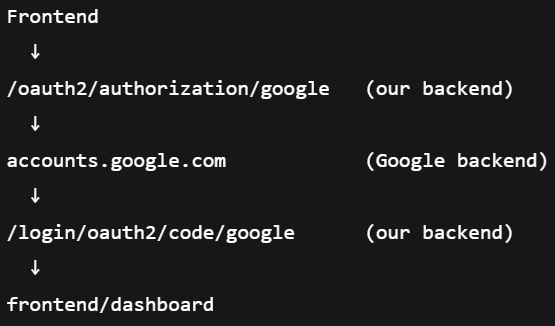
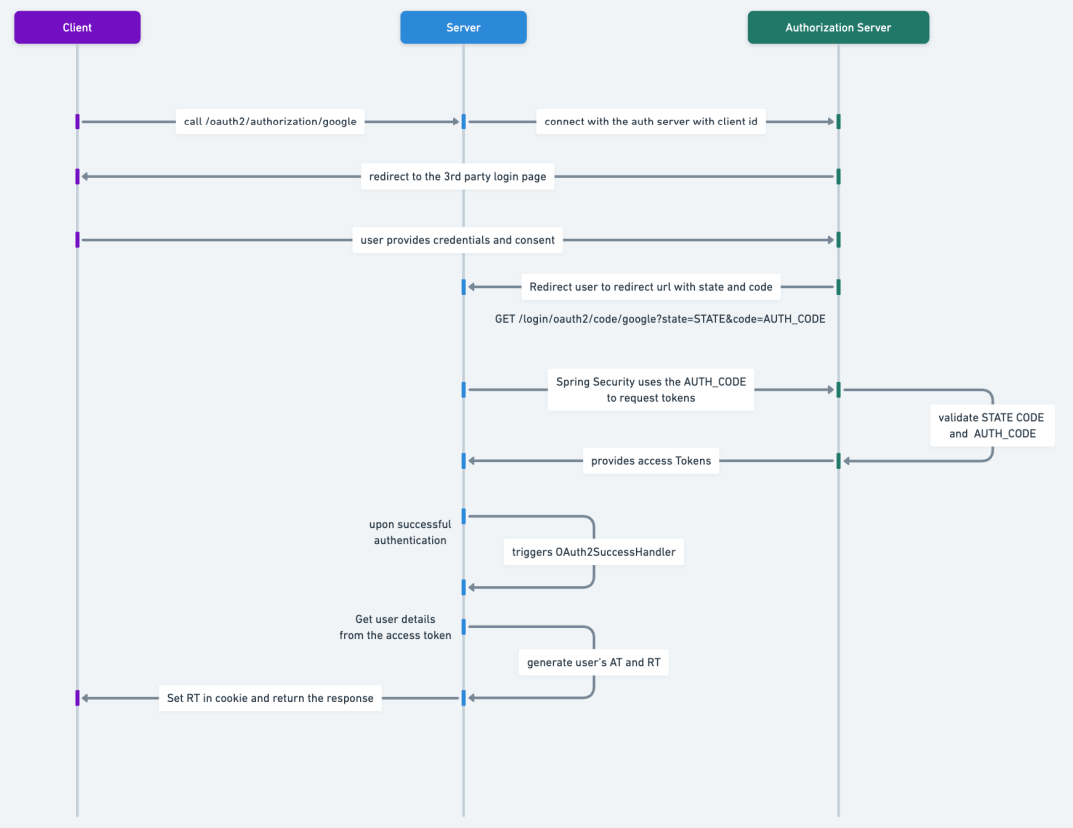
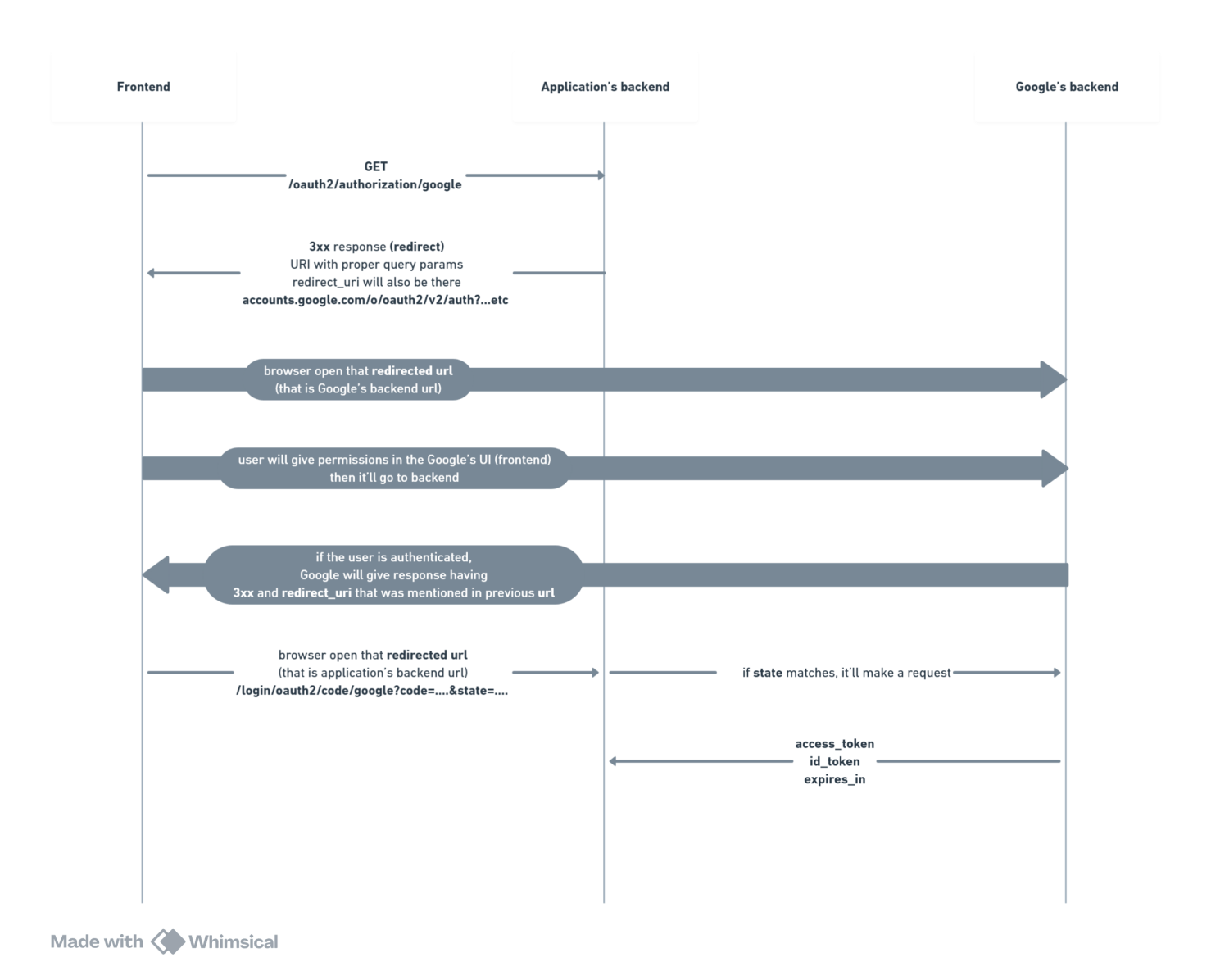
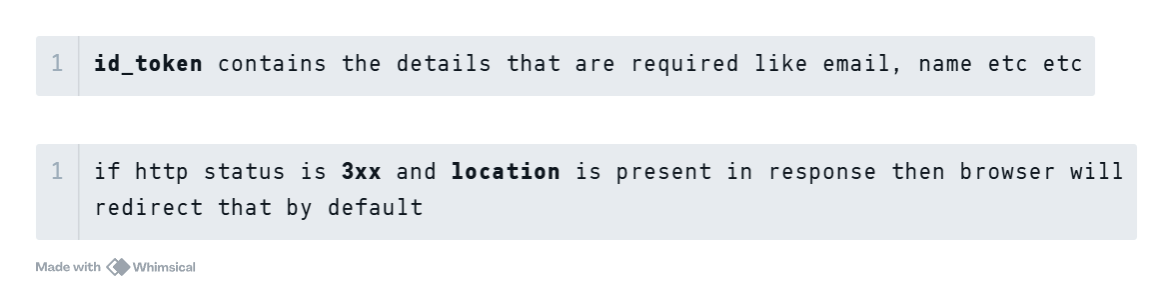
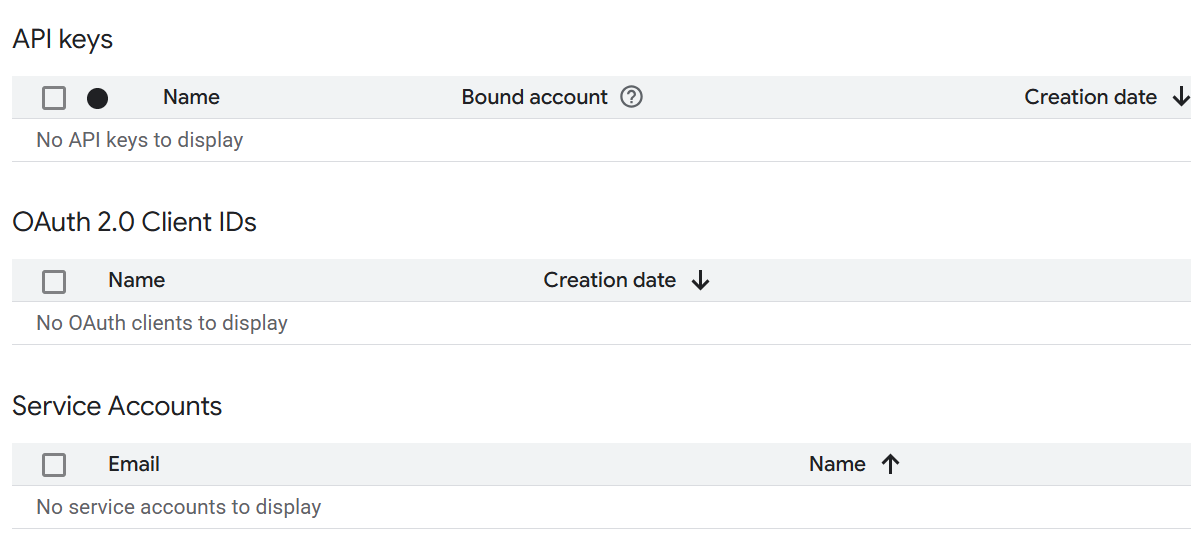
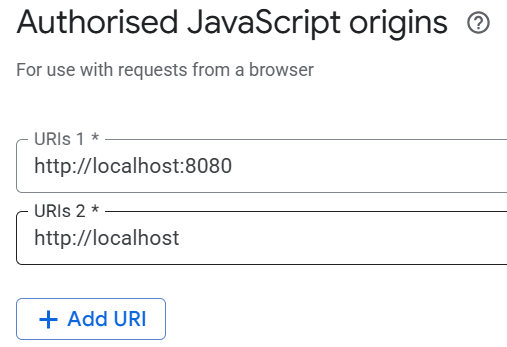
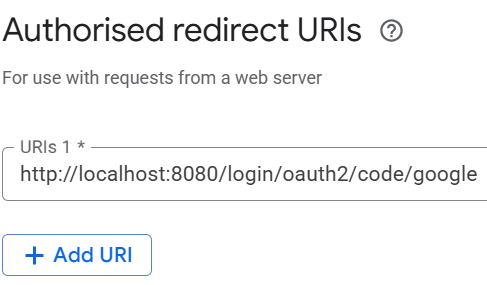
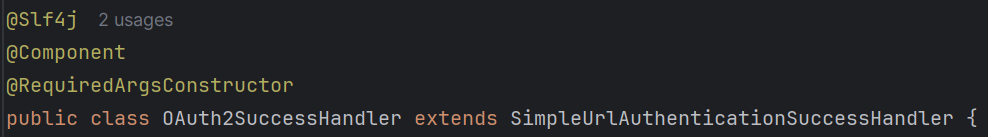
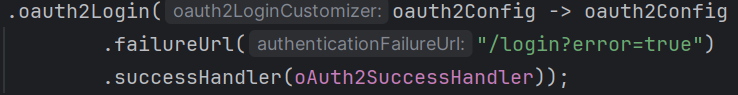
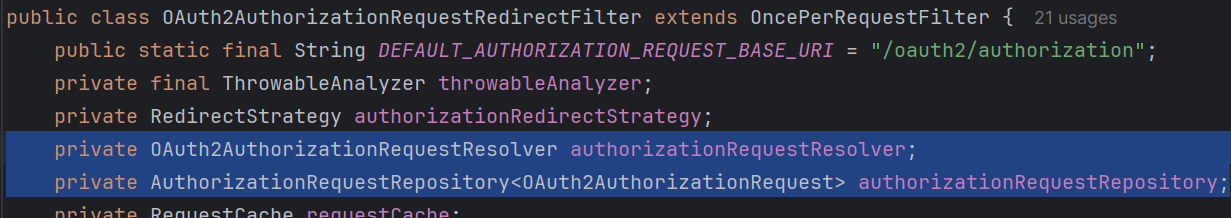
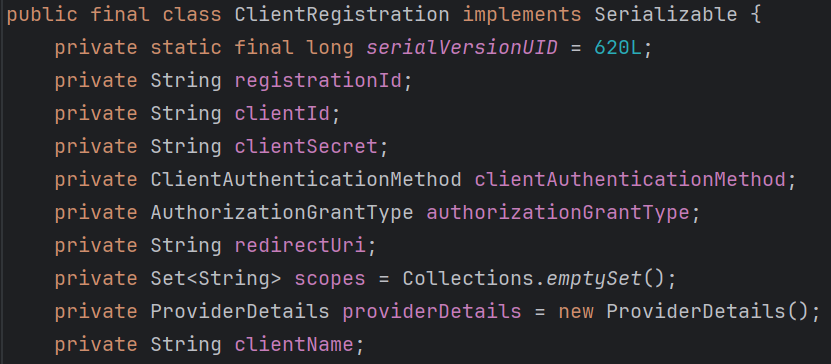
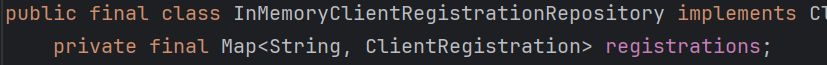
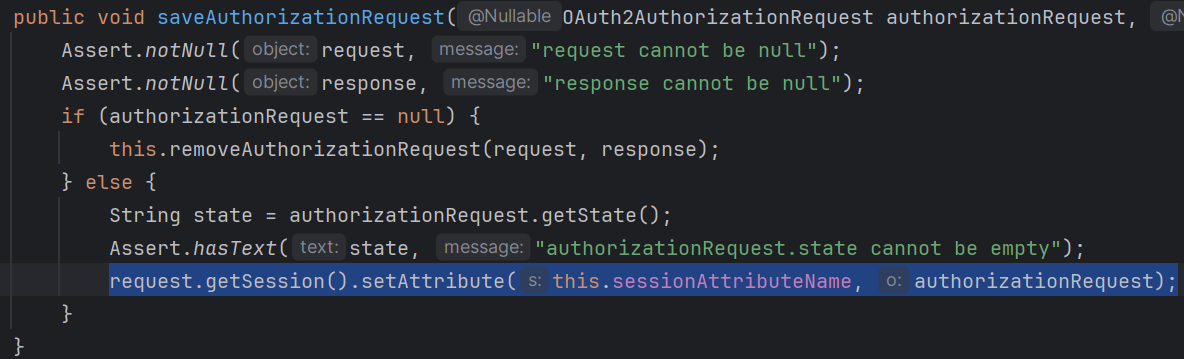
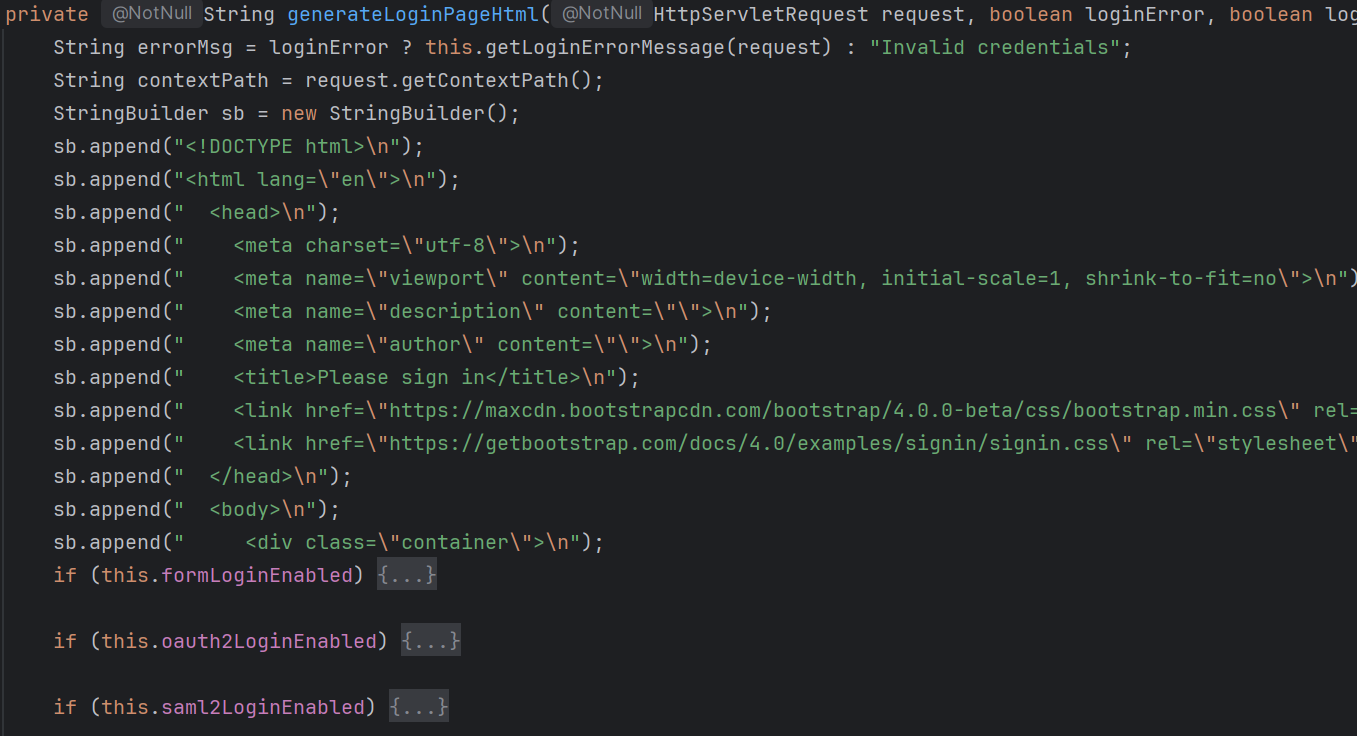
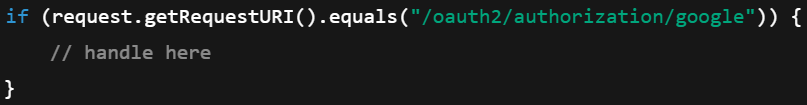
* 2 types of tokens should be there: **access token** (short validity; around 10 min) and **refresh token** (long validity; like 6 months)
  + When the *access token* is expired, the **refresh token** will be used to get a new *access token*.
  + Usually, **refresh token** is stored in the cookies, and 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.
  + 
  + Just keep **user id** in the refresh token, no need to keep everything there.
  + 
  + **Login service method** should get the user from **principal** not from ***database***
  + 
    - 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
  + 
  + 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
  + 
  + Note: **redirect\_uri** is present in this URL.
* Now, our Backend responds with one thing only
  + 
  + 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
  + 
  + 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.
  + 
  + 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**
    - 
  + **id\_token** is a JWT which contains the user details within it like the below
    - 
  + 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.



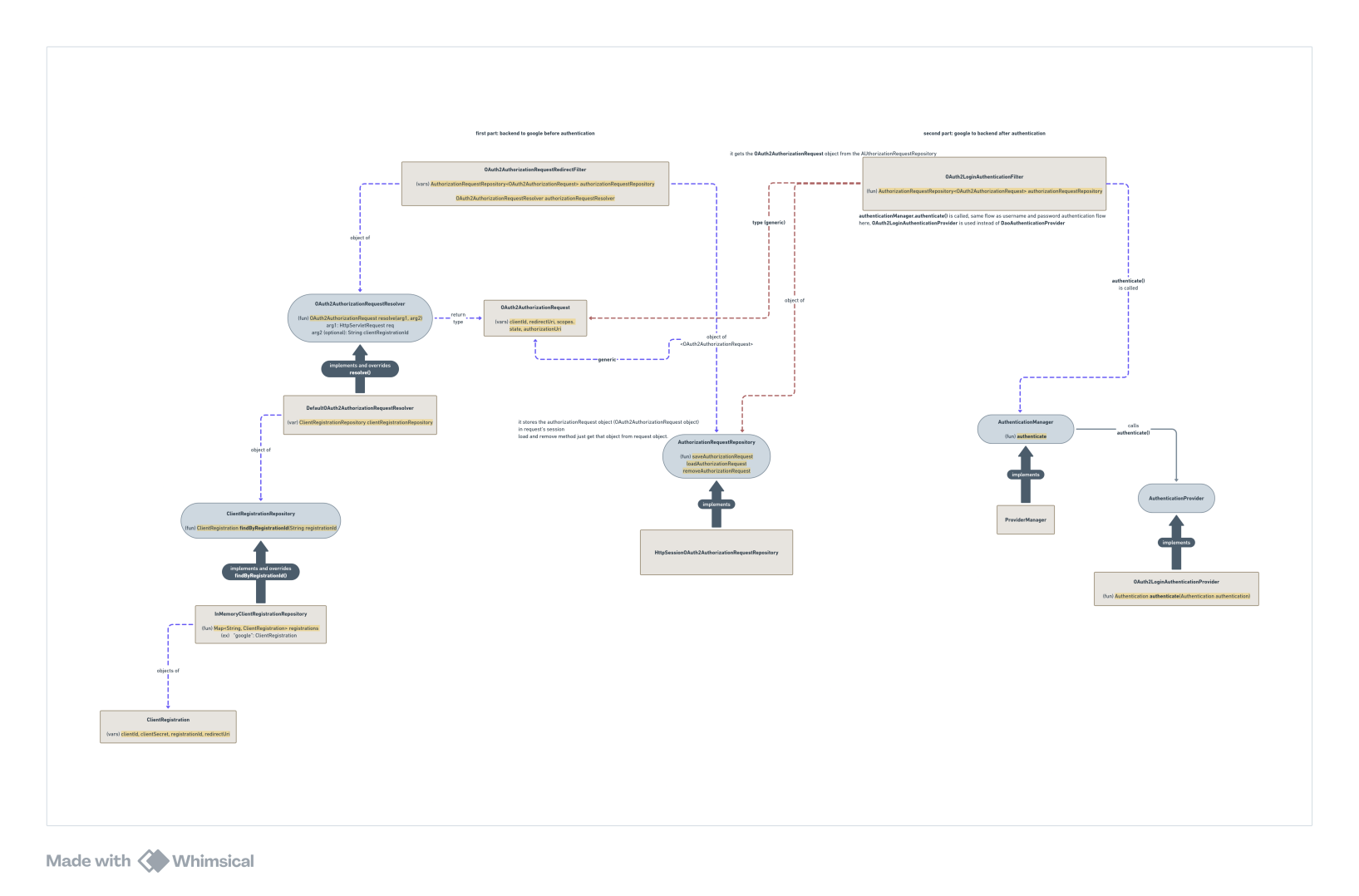


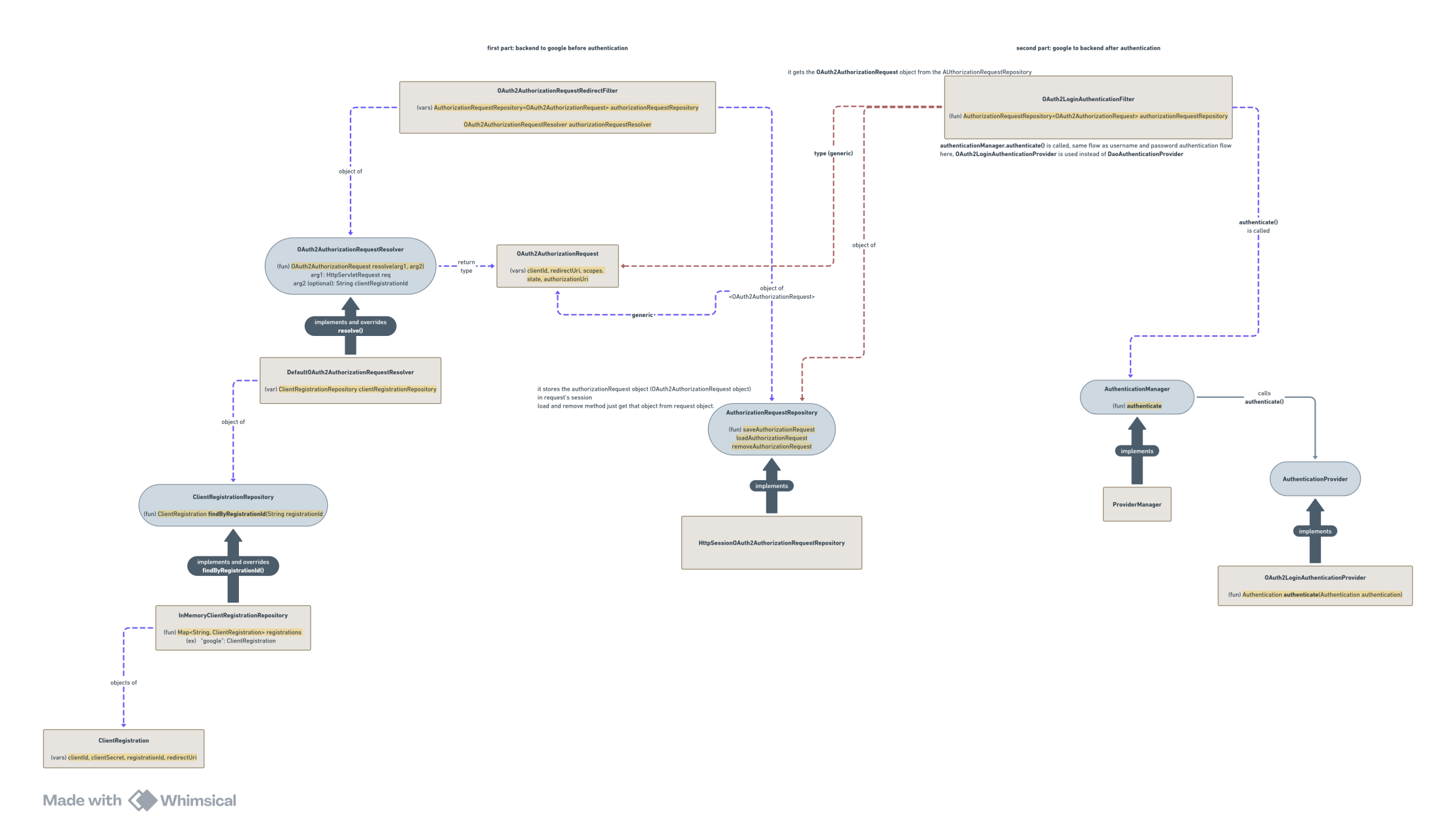


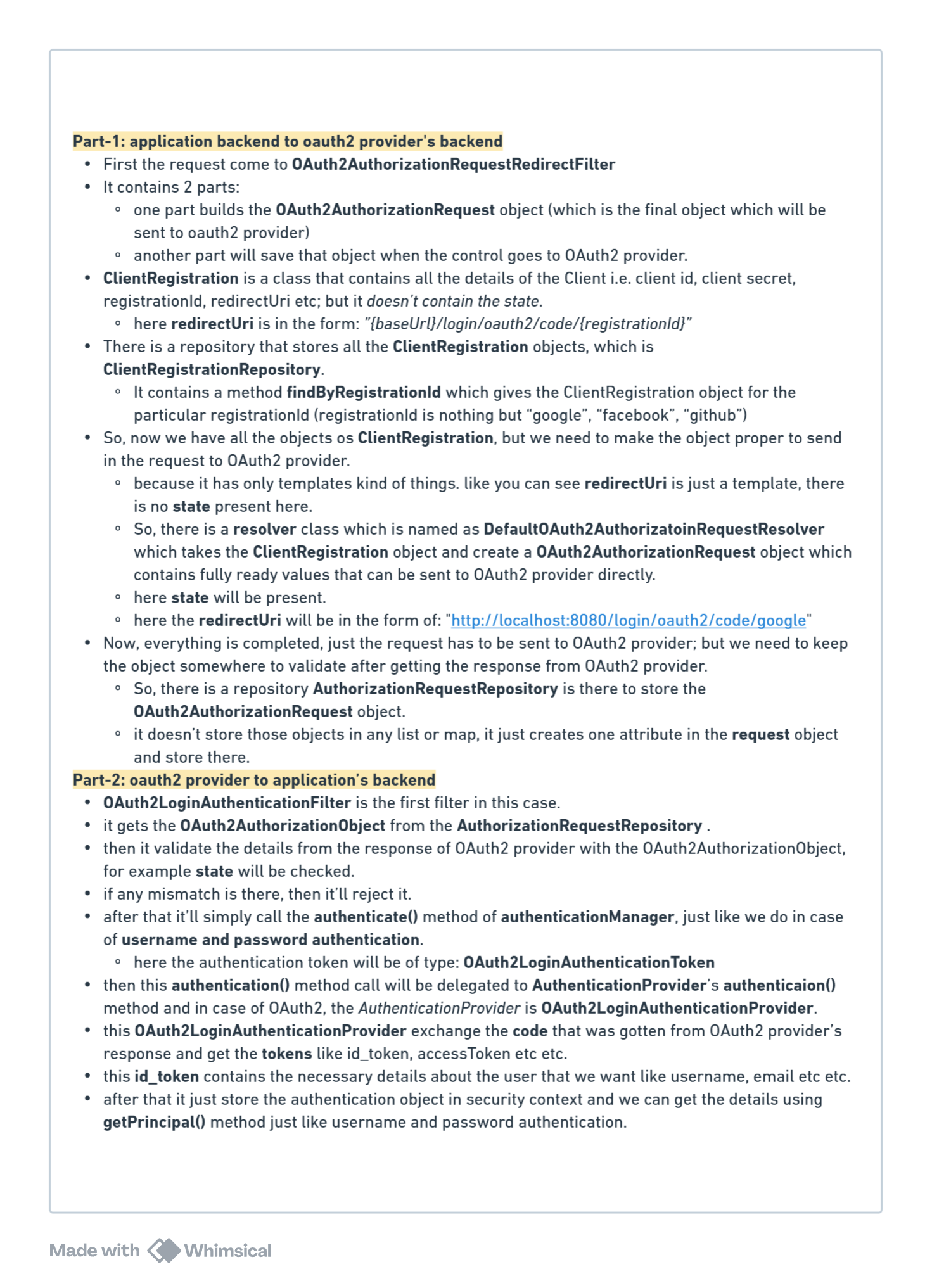


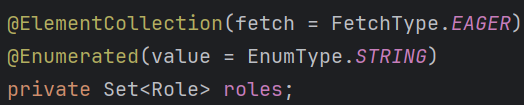
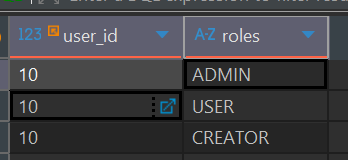
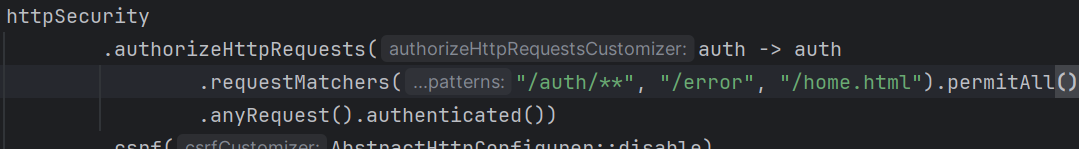
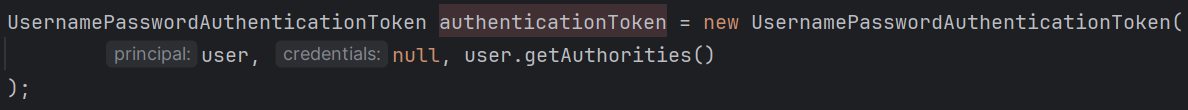
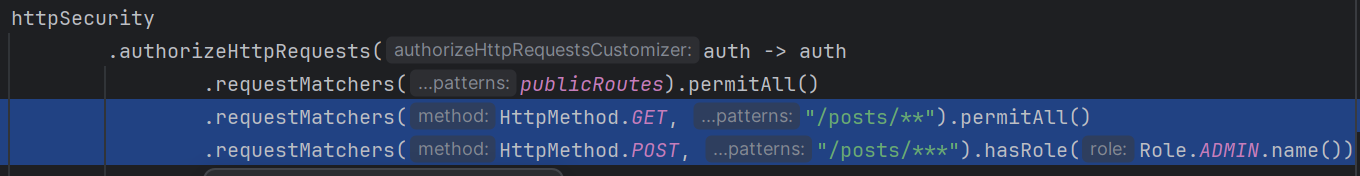
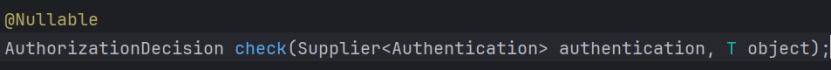
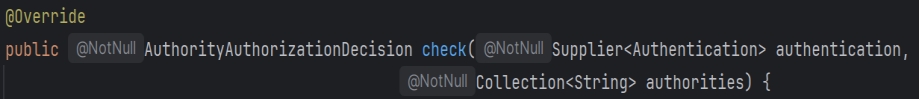
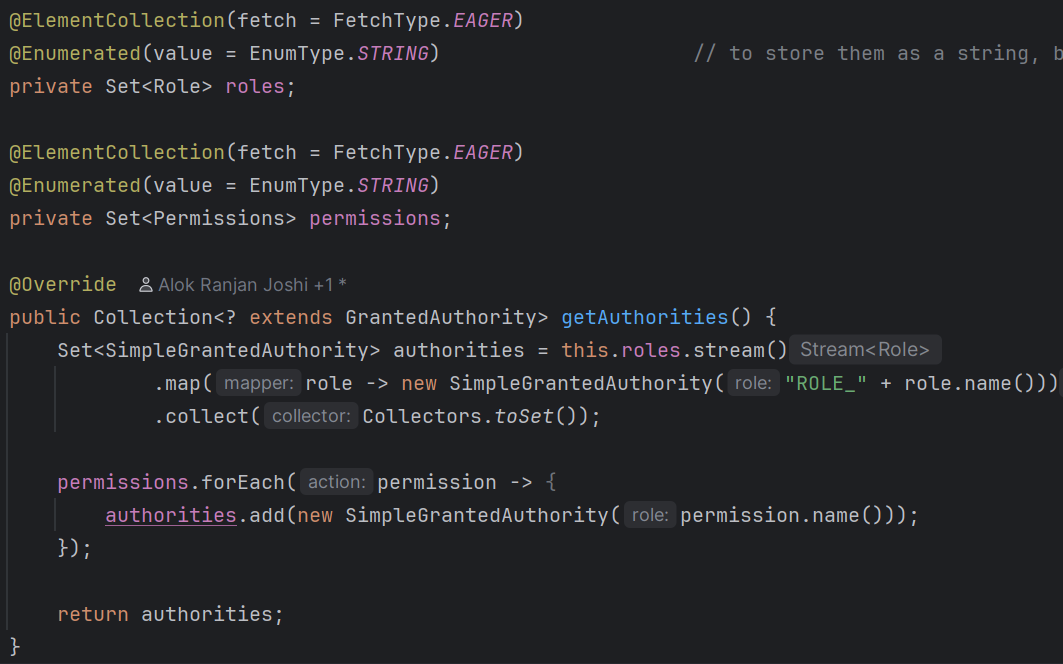
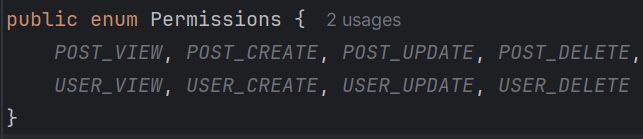
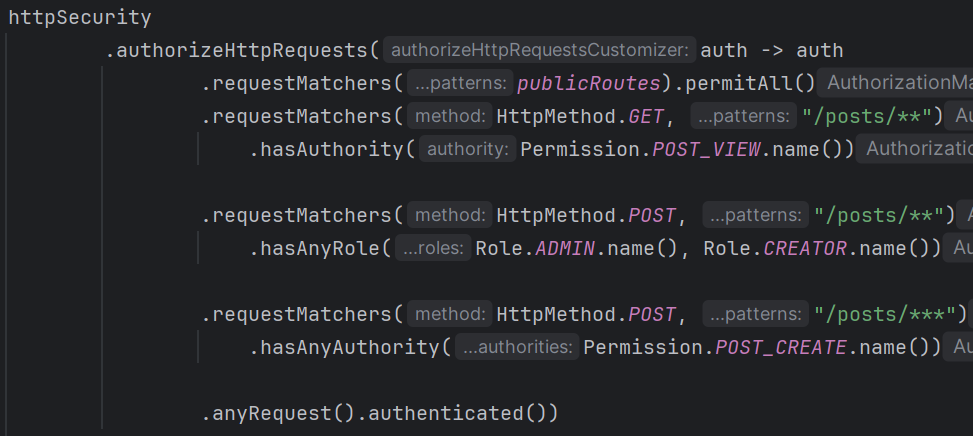
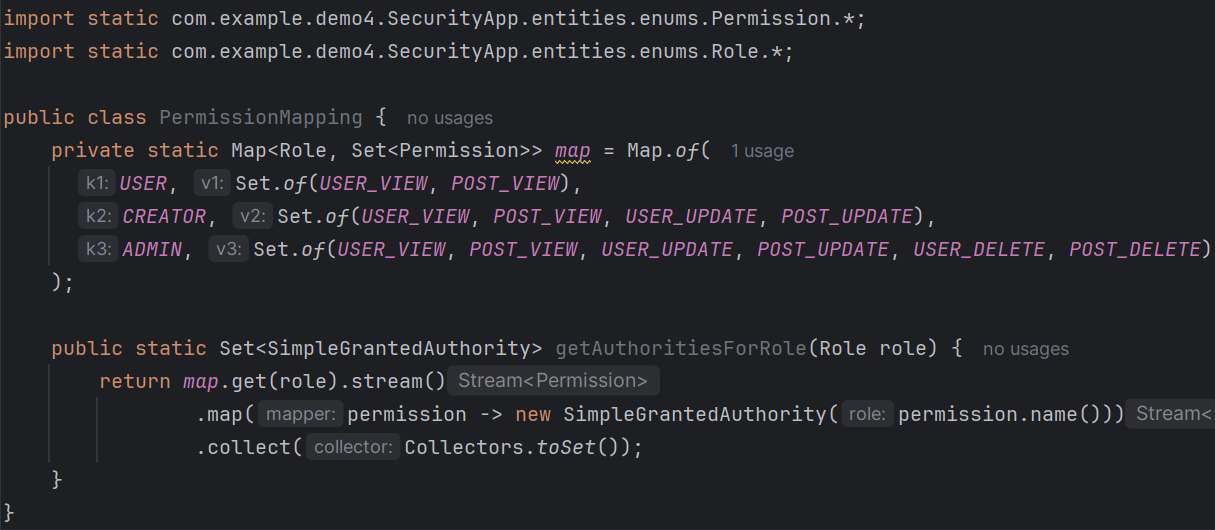
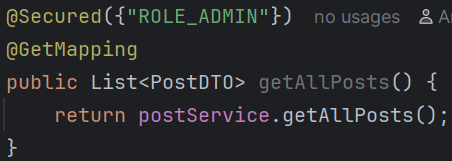
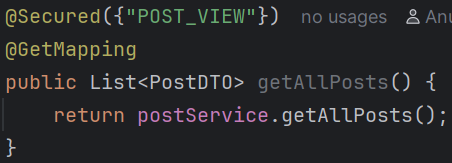
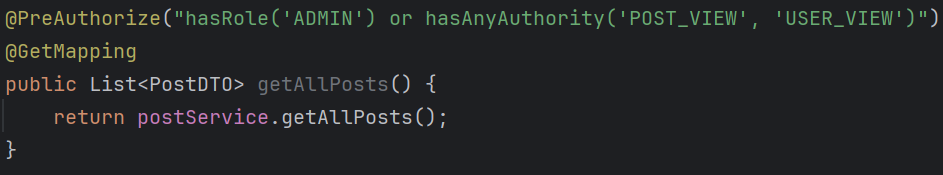
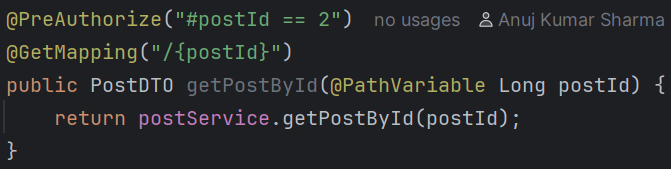
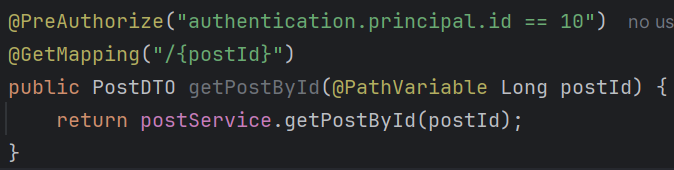
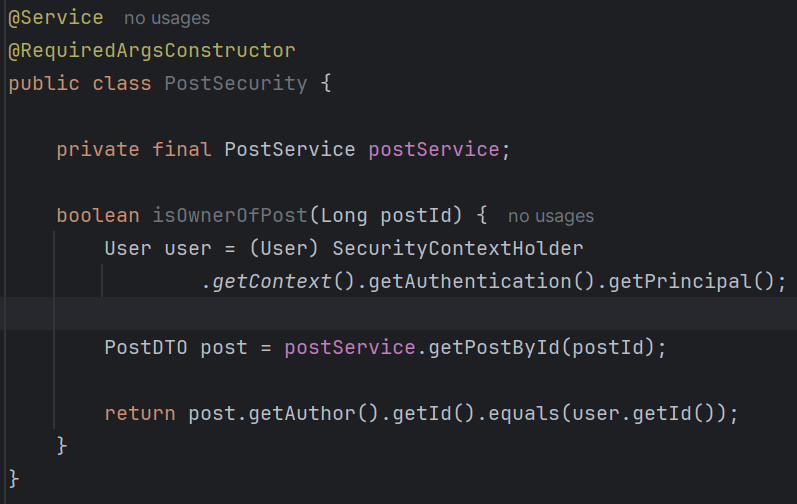
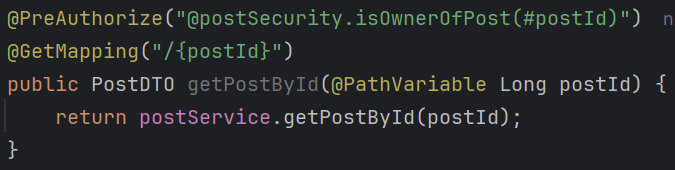
* **Steps**
* **Google Console Setup**
* Select the project in Google Console; and go to Dashboard
* API & Services >> Credentials
  + 
* **Create Credentials** >> **OAuth Client ID**
* **Configure Consent Screen**
  + You need to configure all the things separately.
  + Audience: either you can setup 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;
    - 
      * It means Requests coming from a browser whose origin is “http://localhost:8080” are allowed to start OAuth.
    - (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
  + 
* 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:
    - 
* Now just add that class’s object in the SecurityFilterChain
  + 
* 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.
* 
  + 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**
  + 
* 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}"
    - 
* 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
  + 
* 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.
  + 
* 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.
  + 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 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** .
  + **OAuth2LoginAuthenticatoinProvider** is the one who exchange the **code** to get the **tokens** .
    - It is done with the help of **DefaultAuthorizationCodeTokenResponseClient**
* --------------------------------------------------------------------------------------------------------------
* 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.
  + 
* **NOTE ---------------------------------------------------------------------------------------** 
  + The redirected URIs are matches at the **filter** level, not **servlet level**. Spring Security is completely ***filter based****.*
  + 









* **Authorization**
* **@ElementCollection** 
  + If we write this annotation on a field inside the entity, it’ll create a separate table in database and store those.
  + It is only used in case of collections like **list, set** ..etc.
  + You can say it is like **one-to-many** relation type, but that’s not correct;
    - In case of relation (one-to-many or many-to-one), both the entities are having their own **primary keys**, but in this case, the field that is being stores as a collection is not having any primary key.
  + Relation are created where both the entities are independent and both are having their own primary keys, but **@ElementCollection** just store the values not the new entity.
  + 
  + Here **Role** is just a enum, not an entity and it doesn’t have its own identity.
  + 
  + It is the **user\_roles** table which is created due to **@ElementCollection**, it doesn’t have any primary key; it just map the **user\_id** to a particular value.
  + As I have set **@Enumerated** here for **EnumType.STRING**, so the values are being stored in the database; otherwise the ordinals i.e. 0,1,2,… would have been stored.
* Role is the main concept for **authorization**. Depending upon the role of the user, it’ll be decided that what are the thing he/she can access.
* We can the particular *routes* as authorized for some particular type of users
  + 
    - Here we are simply writing **permitAll** and remaining request as authenticated.
* 
  + Here just return the **SimpleGrantedAuthority** type of object.
  + Because the return type is something that extends **GrantedAuthority** and SimpleGrantedAuthority is the basic implementation of GrantedAuthority.
* In the JWT filter while creating the **UsernamePasswordAuthenticationToken**, make sure to set the **authorities** field, otherwise no user will be having any authorities.
  + Spring Security calls **auth.getAuthorities()** (auth is the Authentication object) and validate the authorities.
  + 
* You can also set authorization for different type of request i.e. GET, POST, ..etc
  + 
* F
* **The flow happens like the below** 
  + **AuthorizationFilter** is the first one that comes in case of authorization.
  + Just like AuthenticationManager, one interface **AuthorizationManager** is also there that contains one method **check()**
    - 
  + **AuthorizationFilter** is having the method **doFilter** which calls this **check()** method of **AuthorizationManager**
    - IMG_257
  + The class **RequestMatcherDelegatingAuthorizationManager** implements **AuthorizationManager** passing the generics as **HttpServletRequest** so it get called.
    - IMG_258
  + After this, **check()** method of this *RequestMatcherDelegatingAuthorizationManager* calls the **check()** method of **AuthoritiesAuthorizationManager** (it is another child class of **AuthorizationManager** and it implements marking the generics as **Collection<String>**) and here the authorities are checked.
    - IMG_259
  + 
    - It gets the **authentication** object and **authorities**
    - Here **authorities** means the authorities required to access the *route*.
    - It’ll fetch the **authorities** from the **authentication** object, and check if any of those authorities is present in the ***required authorities*** list.
* F
* **hasRole** vs **hasAuthorities** 
  + **hasRole(“ADMIN”)** is just a syntactic sugar for **hasAuthorities(“ROLE\_ADMIN”)**
  + If you are storing the roles with the prefix **ROLE\_** then **hasRole()** is handy.
  + For example: ROLE\_USER, ROLE\_ADMIN … are the roles present inside database.
    - Here we can just write **hasRole(“USER”)**  --- it’ll automatically add the prefix ROLE\_ and check i.e. it’ll be checked as **ROLE\_USER**.
    - If you write **hasAuthorities(“USER”)** then it’ll be checked as **USER** only.
* **Role** is kind of top-level authorization like which type of user can access what. **Authority** is bottom level authorization, like more granular authorization.
  + Lets suppose USER, ADMIN are the roles.
  + USER can view the things, you can give it the authorities like VIEW\_POST, VIEW\_USER
  + ADMIN can do everything, so you can give it the authorities like CREATE\_POST, DELETE\_POST etc etc.
  + In simpler terms, **role** is top level permissions; after role **authorities** decide the low level permissions.
* I created one **Authority** enum and implemented the below:
  + 
  + 
* Now you have to add those authorization in the custom filter chain
  + 
  + You can write **hasAuthority()** or **hasAnyAuthority()** and **hasRole()** or **hasAnyRole()** with the same ***requestMatchers()***, you need to write one ***requestMathcer()*** for ***role*** or ***authority***.
  + You can give multiple authorities like this way.
* Otherwise you can create a hard-coded maps of **roles** with **permissions** like some particular role user can have some particular set of permissions.
  + 
  + NOTE: You can import **enum values** directly using **import static**
  + Now you don’t need to store the ***permissions*** in the database.
  + 
* As we can see, writing this many logic in the **WebSecurityConfig** is creating overhead and messy.
  + In case of real world project, there will be so many paths, roles, permissions and if we write all those authorization then it’ll be very complicated.
  + Therefore **@Secured** and **@PreAuthorize** and **@PostAuthorize** are used.
  + These are used for **method level authorization**, we just need to write these with the methods.
  + **@EnableMethodSecurity** is required to use these annotations. Just write once on any **Component** class (you can write with any class having *@Configuration* or *@Component*)
  + For using **@Secured**, you need to write
    - 
  + For using **@PreAuthorize** or **@PostAuthorize**, you need to write
    - 
    - By default **prePostEnabled** is **true** only, so even if you don’t mention this explicitly, it’ll work.
* **@Secured**
  + It’ll only work for **roles**, i.e. having **ROLE\_** prefix.
  + It’ll not work for any other permissions like POST\_VIEW, POST\_DELETE etc etc.
  + **ROLE\_** must be there otherwise it’ll not work.
  + 
    - Now the users having roles other than **ADMIN** will not be able to access this.
  + NOTE: You need to write **ROLE\_ADMIN** here, only **ADMIN** will not work.
  + 
    - I wrote this POST\_VIEW and tried to access the posts from the user who has POST\_VIEW permission. But it didn’t work.
  + So, permissions having prefix **ROLE\_** will only work in case of **@Secured**
* **@PreAuthorized** 
  + It is widely used; @Secured is used very rarely (negligible)
  + It can access the **arguments of the method**.
  + It supports **role**, **permissions**, **conditions** (and, or etc), **parameters** (method arguments)
  + You can write **hasRole**, **hasAnyRole**, **hasAuthority**, **hasAnyAuthority** …etc etc inside it.
  + 
    - We can write the query like this. You can use **and** or **or** and write multiple filters (authorizing filters; not spring filters…) here.
  + 
    - You can also access the method argumentsusing **#**
    - In here I accessed *postId* using **#postId**
  + Also you can access **authentication** object inside **@PreAuthorize**.
    - 
    - NOTE: here don’t make getter function call;
    - Just write **authentication.principal.id** or something like that.
    - But, if the **authentication** is null, then it’ll fail ***silently***.
* You can also create Security service for authorization; and it is recommended.
  + I want to display the post that was searched by **postId**, only if the user is the author of this post.
  + 
    - Created one service **PostSecurity** and wrote this method inside that.
  + 
    - The bean of **PostSecurity** can be accessed by **@postSecurity** inside the **@PreAuthorize** , and the argument *postId* can be accessed as **#postId**
    - **p** will be ***small*** in **@postSecurity** because the bean name will be **postSecurity** by default.
* You can write **@Secured**, **@PreAuthorize, @PostAuthorize** anywhere; but it is the best practice to use those at **controller** level; because in case of unauthorized condition, the control should not even go to the method.
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F