Creazione app backend su eclipse utilizzando Spring-boot 3.3, progetto di tipo maven e usando Java 17, implemento le seguenti dipendenze alla creazione:

- Spring Boot DevTools
- Spring Data JPA (gestione Database)
- PostgreSQL Driver (gestione Database
- Spring Web
- Spring Security
- OAuth2 Client
- OAuth2 Resource Server
- Azure Active Directory

Una volta generata l'app per gestire lo Scaffholding al suo interno creerò i seguenti package:

- config (gestirà il securityFilterChain, webClient e tutto ciò che sarà necessario per permettere gli accessi alle api)
- entities (dove creeremo le entità necessarie che utilizzeremo nel progetto)
- dtos (le classi di supporto per le entità)
- controllers (dove creeremo le api da utilizzare con il frontend
- repositories (dove implementeremo JpaRepository per fare le ricerche nel DB
- services (per creare tutte le funzioni necessarie di salvataggio/ricerca

Implementando le seguenti librerie sarà necessario creare un DB da collegare alla nostra app tramite il file application.properties

```
spring.datasource.url=jdbc:postgresql://localhost:ilvostrovalore/nomeDB
spring.datasource.username=nome_Assegnato
spring.datasource.password=password_Assegnata


spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql=false
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
```

Dato che ogni provider ritorna informazioni diverse creeremo un'entità UserInfo che salverà le informazioni principali che ci interessano degli utenti:
nome, cognome, immagine profilo, email ed il provider con il quale ha fatto l'accesso e l'id

```
@Table(name = "users")
@Entity
```

```java
public class User{
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "user_id")
  private Long id;
        @Column(name = "name")
        private String name;
        @Column(name = "surname")
        private String surname;
        @Column(name = "email")
        private String email;
        @Column(name = "provider")
        private String provider;
        public String getName() {
                return name;
        }
        public Long getId() {
                return id;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getSurname() {
                return surname;
        }
        public void setSurname(String surname) {
                this.surname = surname;
        }
        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
        public String getProvider() {
                return provider;
        }
        public void setProvider(String provider) {
                this.provider = provider;
        }
}
```

e creiamo un UserDTO che ci servirà per ottenere più facilmente le informazioni degli utenti più avanti

```
public class UserDTO{
	private String name;
	private String surname;
	private String email;
	private String provider;
/*qui creiamo un costruttore che ci servirà in seguito, al momento ancora non abbiamo creato il GoogleUserDTO ma ci servirà tra poco*/

	public UserDTO(GoogleUserDTO dto) {
		this.name= dto.getGiven_name();
		this.surname= dto.getFamily_name();
		this.email= dto.getEmail();
	}
/*per ogni provider che implementeremo creeremo nuovi costruttori cosi da poterli implementare, qui sotto potete vedere come saranno i costruttori che ci serviranno per Azure e per Facebook
	public UserDTO(AzureUserDTO dto) {
		this.name = dto.getGivenName();
		this.surname = dto.getFamilyName();
		this.picture = dto.getPicture();
		this.email = dto.getEmail();
	}

	public UserDTO(FacebookUserDTO dto) {
		String[] nameSurname = dto.getNameParts();
		this.name = nameSurname[0];
		this.surname = nameSurname[1];
		this.picture = dto.getPictureUrl();
		this.email = dto.getEmail();
	}



*/
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getSurname() {
		return surname;
```

```
        }
        public void setSurname(String surname) {
                this.surname = surname;
        }
        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
        public String getProvider() {
                return provider;
        }
        public void setProvider(String provider) {
                this.provider = provider;
        }
}
```

Adesso dovremo configurare la Repository degli utenti per la ricerca nel DB tramite sia solo attraverso l'username che attraverso Username e Provider

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
  Optional<User>User findByEmail(String email);


}
```

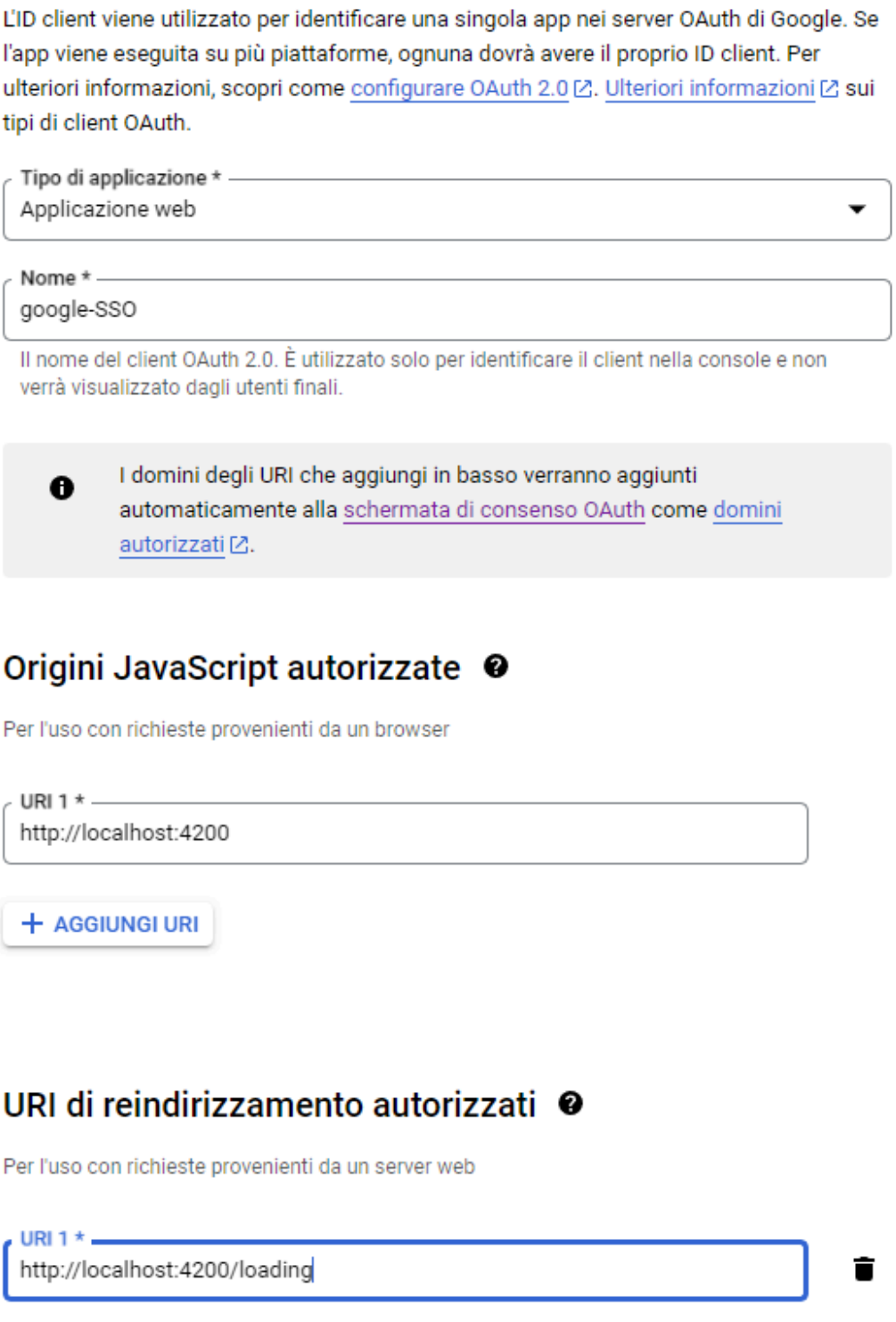
Ed ora creiamo UserService che ci servirà per andare a salvare gli utenti nel caso accedano per la prima volta e verificare che un utente non possa accedere tramite altri provider

```
@Service
public class  UserService {
        @Autowired
  private UserRepository userRepository;
  private final BCryptPasswordEncoder passwordEncoder = new
BCryptPasswordEncoder();

  public String registerAccount(UserDTO userDTO) throws Exception {
     System.out.print("sono prima di validate account\n");
     //validate data from client
     validateAccount(userDTO);
     System.out.print("sono dopo di validate account\n");
     User user = insertUser(userDTO);
```

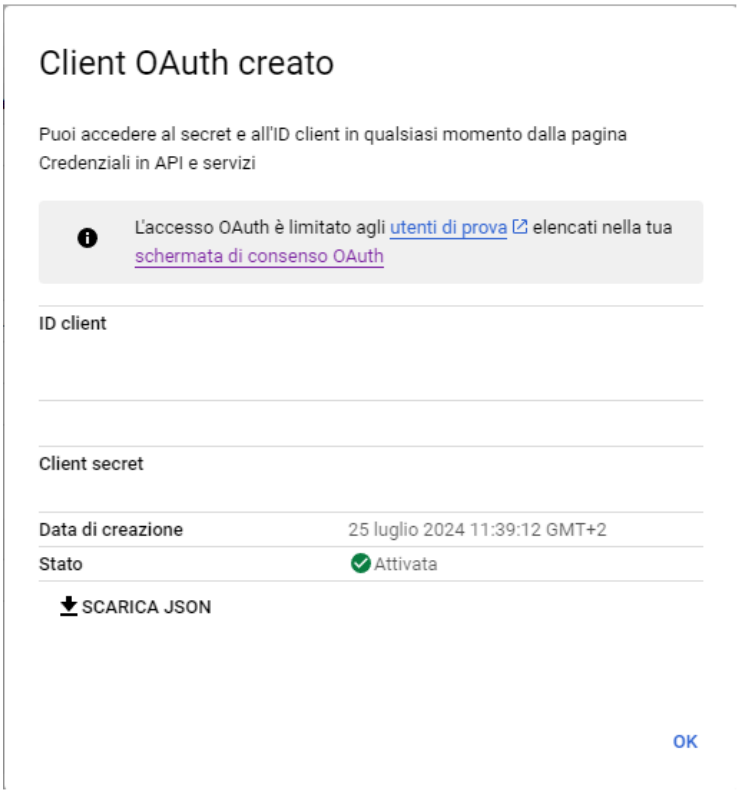
```
        String response;
        try {
            userRepository.save(user);

            response = "Register account  successfully!";
        }catch (Exception e){
            response = "Service Unavailable";
            //throw new
BaseException(String.valueOf(HttpStatus.SERVICE_UNAVAILABLE.value()),
"Service Unavailable");
        }
        return response;
    }



    private User insertUser(UserDTO userDTO) {
        User user = new User();
        user.setEmail(userDTO.getEmail());
        user.setName(userDTO.getName());
        user.setProvider(userDTO.getProvider());
        return user;
    }
    private void validateAccount(UserDTO userDTO) throws Exception{
        if(ObjectUtils.isEmpty(userDTO)){
            System.out.print("user is empty\n");
            throw new Exception();
        }
      User user = userRepository.findByEmail(userDTO.getEmail()).get();
        if(!ObjectUtils.isEmpty(user)){
            System.out.print("user already exist\n");
            throw new Exception();
        }
    }
```

Ora che la base è creata passiamo alla creazione di un'app google che ci servirà per poter implementare l'autentificazione SSO del provider
quindi per prima cosa andare sul seguente link:
https://console.cloud.google.com/welcome
accedere con un account sviluppatore (se non si è in possesso di un account simile è possibile crearlo gratuitamente).
Una volta fatto l'accesso
andare su:

API e Servizi → credenziali → crea credenziali: Scegliere Id client Oauth ed impostarlo in modo simili al seguente:



ed appena creato vedremo:



salvate il client Id e Secret che vedete ed andate a salvarlo dentro il file application.properties

```
google.client-id=your_client_id
google.client-secret=your_client_secret
google.scope=email,profile,openid
```

prima di iniziare la configurazione del controller ci servirà creare una classe di supporto per ottenere le informazioni che si ottengono da google per poi poterle inserire nell'utente finale che useremo

ora andiamo a configurare il file dove creeremo gli endpoint da utilizzare nel front-end che chiameremo GoogleController, qui dovremo creare un endpoint dove ritorneremo un url che andremo a buildare con la configurazione di google che cliccando su di esso manderà l'utente su una pagina per l'accesso e poi se eseguito correttamente lo reindirizzerà nella pagina che avremo scelto noi, poi ci faremo ritornare un token di accesso che ci servirà per poter avere l'autentificazione che è andato tutto a buon fine e che potremo entrare nella pagina privata ed accedere ai nostri dati privati

per prima cosa dovremo importare nel nostro file pom le seguente dipendenze, entrambe servono per poter utilizzare webClient per quando dobbiamo buildare gli url e le richieste per ottenere token di accesso e le informazioni degli utenti

```
        <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-webflux</artifactId>
        </dependency>
        <dependency>
                        <groupId>com.google.api-client</groupId>
                        <artifactId>google-api-client</artifactId>
                         <version>2.2.0</version>
        </dependency>
```

quindi ora creeremo GoogleController per gestire come ricevere l'url per andare ad accedere, creeremo una funzione per ottenere il token di accesso per poter entrare nelle pagine private che creeremo nel frontend e una funzione per gestire il salvataggio dell'utente nel DB

```
@RequestMapping("/google")
public class GoogleController {

        @Value("${google.client-id}")
        private String clientId;
        @Value("${google.client-secret}")
        private String clientSecret;
        private static final String REDIRECT_URI = "http://localhost:4200/loading";

        @Autowired
        private UserService userService;

        @GetMapping("/url")
        public ResponseEntity<String> auth(){
        /*normalmente dovremmo costruire manualmente l'url indicando cosa sia ogni cosa, ma google ci permette di abbreviare la procedura grazie alla dipendenza aggiunta*/
                String url = new GoogleAuthorizationCodeRequestUrl(
                                clientId,
                                REDIRECT_URI,
                                Arrays.asList("profile","email")
                                ).build();
                return ResponseEntity.ok(url);
        }

  @GetMapping("/getToken")
  public ResponseEntity<String> getToken(@RequestParam("codeValue") String codeValue) {
```

```
      try {
         /*non ho usato la libreria di google perché causava problemi nel building della
richiesta per ottenere il token, inoltre questo codice è più facile da rendere generico
in modo da poterlo riutilizzare per più provider*/
         String tokenResponse = webClient.post()
                 .uri("https://oauth2.googleapis.com/token")
                 .header(HttpHeaders.CONTENT_TYPE,
MediaType.APPLICATION_FORM_URLENCODED_VALUE)
                 .bodyValue("code=" + codeValue +
                         "&client_id=" + clientId +
                         "&client_secret=" + clientSecret +
                         "&redirect_uri=" + REDIRECT_URI +
                         "&grant_type=authorization_code")
                 .retrieve()
                 .bodyToMono(String.class)
                 .block();
         return ResponseEntity.ok(tokenResponse);
      } catch (Exception e) {
         System.out.print("\nsiamo dentro l'errore\n" + e.getMessage());
         return
ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(e.getMessage());
      }
   }


         private UserDTO findUserInDB(String accessToken) throws Exception
         {
/*qui invece usiamo il nostro web client per assemblare l'url alla quale verrà mandata
la richiesta per ottenere le nostre informazioni,  inoltre stiamo mappando l'utente
google dentro l'utente finale che vorremo visualizzare, questa logica la utilizzeremo
per ogni provider creando un dto apposito del provider e poi andando a salvarlo
creando un costruttore apposito dentro userDTO*/
                  UserDTO userInfo = new UserDTO(webClient.get()
            .uri(uriBuilder -> uriBuilder.path("/oauth2/v3/userinfo")
                    .queryParam("access_token", accessToken)
                    .build())
            .retrieve()
            .bodyToMono(GoogleUserDTO.class)
            .block());
                  System.out.print("\n\nemail:"+userInfo.getEmail()+"\n\n");
                  userInfo.setProvider("google");
                  Optional<User> user = userService.findByEmail(userInfo.getEmail());
                  if(user.isEmpty())
                  {
```

```
                        String response = userService.registerAccount(userInfo);
                }
                return userInfo;
        }

        private final WebClient webClient;
          public GoogleController(WebClient.Builder webClientBuilder) {
              this.webClient =
webClientBuilder.baseUrl("https://www.googleapis.com").build();
//url base per il richiamo delle api di google
          }


         @GetMapping("/userInfo")
           public ResponseEntity<UserDTO>
getUserInfo(@RequestParam("accessToken") String accessToken) {
              try {
                  UserDTO user = findUserInDB(accessToken);
                  return ResponseEntity.ok(user);
              } catch (Exception e) {
                  return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
              }
          }
}
```

ora se provassimo a creare il frontend non potremmo avere accesso alle api perché in realtà non abbiamo ancora configurato ciò che ci permette di vedere le API, dovremo creare un file WebConfig con il quale daremo il permesso di accedere ai nostri endpoint dall'url del frontend e per non farci negare l'accesso dalla policy CORS

```
@Configuration
@EnableWebMvc
public class WebConfig {

  private static final Long MAX_AGE = 3600L;
  @Bean
  public CorsFilter corsFilter() {
      UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
      CorsConfiguration config = new CorsConfiguration();
      config.setAllowCredentials(true);
      config.addAllowedOrigin("http://localhost:4200");
      config.setAllowedHeaders(Arrays.asList(
              HttpHeaders.AUTHORIZATION,
```

```
            HttpHeaders.CONTENT_TYPE,
            HttpHeaders.ACCEPT));
    config.setAllowedMethods(Arrays.asList(
            HttpMethod.GET.name(),
            HttpMethod.POST.name(),
            HttpMethod.PUT.name(),
            HttpMethod.DELETE.name()));
    config.setMaxAge(MAX_AGE);

    source.registerCorsConfiguration("/**", config);
    CorsFilter bean = new CorsFilter(source);
    return bean;
  }
}
```

ora ho creato velocemente il file AuthConfig per permettere l'accesso ai nostri endpoint e gestire

```
@Configuration
@EnableWebSecurity
public class AuthConfig {

  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      http
            .authorizeHttpRequests(auth->auth
                  .requestMatchers("/google/**").permitAll()
                  .anyRequest().authenticated())
                .oauth2Login(oauth2Login ->//questa parte va a configurare il login di
tipo oauth durante la creazione,
            oauth2Login
                // Pagina di login, se necessaria
                .successHandler(new
SimpleUrlAuthenticationSuccessHandler("/loading")) // Redirigi su /loading dopo il
login
                .failureHandler((request, response, exception) -> {
                    response.sendRedirect("/");
                })
        )

            .logout(log -> log
                  .logoutRequestMatcher(new AntPathRequestMatcher("/logout",
"POST"))
                  .logoutSuccessUrl("/")
                  .invalidateHttpSession(true)
```

```
                .deleteCookies("JSESSIONID")
                .clearAuthentication(true)
            )
        .exceptionHandling(customizer->customizer.authenticationEntryPoint(new
HttpStatusEntryPoint(HttpStatus.UNAUTHORIZED)))
        .csrf(conf->
                    conf.disable())
        .httpBasic(httpBasic -> {})

        .exceptionHandling(customizer->customizer.authenticationEntryPoint(new
HttpStatusEntryPoint(HttpStatus.NOT_IMPLEMENTED)))

        ;
    return http.build();
  }


}
```

creazione lato frontend

nel nostro caso stiamo usando angular 18 ma utilizzando la procedura lazy-loading per il routing (quindi con i moduli) ed il server-side-rendering (quindi per usare local-Storage dovremo aggiungere dei passaggi. Per la libreria grafica usiamo le varie librerie di prime (primeng, primeflex), trovate la documentazione per usarle qui:
www.primeng.org
www.primeflex.org

quindi se volete fare il frontend come in questa guida i passaggi da fare sono i seguenti

- creazione angular app nel vostro spazio lavorativo dove volete salvare l'app tramite terminale:
  ng new nome-app –no-standalone
  →per la grafica scss (sass)
  →Server-Side-Rendering (y)
- una volta generata l'app installare la libreria grafica:
  npm install primeng
  npm instal primeflex
  npm install primeicons
  per utilizzare primeflex e le icone fornite dalle librerie andare nel file style.scss ed importare:

```
@import '../node_modules/primeflex/primeflex.scss';
@import '../node_modules/primeflex/primeflex.css';
@import '../node_modules/primeflex/primeflex.min.css';

@import '../node_modules/primeicons/primeicons.css';
```

```
@import '../node_modules/primeflex/themes/primeone-light.scss';
//@import '../node_modules/primeflex/themes/primeone-dark.scss'
```

- creiamo i componenti che ci serviranno, in questo caso io ne uso 3:
  -home: dove troveremo i link per effettuare il login
  -loading: dove verremo reindirizzati una volta effettuato il login per poter ottenere i token di accesso
  -private: dove vedremo le informazioni dell'utente
  il comando da terminale è ng generate module "nome-componente" –route "nome-componente" –module app.module
- dentro app-routing.module aggiungere che al path '' si carichi la pagina di home
- cancellare ciò che è dentro app.component.html e sostituirlo con `<router-outlet>` cosi che il routing sarà correttamente implementato
- creare ora una cartella services dove creiamo i file che ci serviranno in ordine a:
  -MyHttpClientService richiamare gli endpoint
  -AuthService: verificare i token
  -AuthGuard: dare l'accesso alle pagine private
  -local-storage: sarà una cartella dentro la quale gestiremo 3 file:

local-storage-token.ts ci servirà per utilizzare le funzioni di window.localStorage altrimenti non disponibili a causa del Server-Side-Rendering:

```
import { InjectionToken } from '@angular/core';
export const LOCAL_STORAGE = new InjectionToken<Storage>('Local
Storage', {
  providedIn: 'root',
  factory: () => typeof window !== 'undefined' ? window.localStorage :
new NoopStorage()
});

class NoopStorage implements Storage {
  length: number = 0;
  clear(): void { }
  getItem(key: string): string | null { return null; }
  key(index: number): string | null { return null; }
  removeItem(key: string): void { }
  setItem(key: string, value: string): void { }
}
```

- Server-Local-Storage-Service, che servirà nel caso il servizio è offline:

```
@Injectable({
  providedIn: 'root',
})
export class ServerLocalStorageService {
  getItem(key: string): string | null {
```

```
    return null;
  }

  setItem(key: string, value: string): void { }

  removeItem(key: string): void { }

  clear(): void { }
}
```

- Browser-Local-Storage-Service, nel caso il servizio è online

```
import { Injectable, Inject } from '@angular/core';
import { LOCAL_STORAGE } from './local-storage.token';

@Injectable({
  providedIn: 'root',
})
export class BrowserLocalStorageService {
  constructor(@Inject(LOCAL_STORAGE) private localStorage:
Storage) {}

  getItem(key: string): string | null {
    return this.localStorage.getItem(key);
  }

  setItem(key: string, value: string): void {
    this.localStorage.setItem(key, value);
  }

  removeItem(key: string): void {
    this.localStorage.removeItem(key);
  }

  clear(): void {
    this.localStorage.clear();
  }
}
```

- ora andare ad importarlo nei providers dell'app.module.ts insieme ad AuthGuard, providerHttpClient(withFetch()) //senza questi import il vostro programma andrà incontro a vari problemi, quindi andate ad impostare

```
providers:[
AuthGuard,
```

```
    provideClientHydration(),
    provideHttpClient(withFetch()),
    {
      provide: LOCAL_STORAGE,
      useFactory: (platformId: Object) => {
        if (isPlatformBrowser(platformId)) {
          return new
BrowserLocalStorageService(window.localStorage);
        } else {
          return new ServerLocalStorageService();
        }
      },
      deps: [PLATFORM_ID]
    }
]
```

ora ancora AuthGuard non è stato impostato quindi potrebbe segnarlo errato ma prima di quello andiamo ad impostare ora il nostro httpClientService per poter gestire gli endpoint

```
@Injectable({
  providedIn: 'root',
})
export class MyHttpClientService {
  constructor(
    private http: HttpClient,
    @Inject(LOCAL_STORAGE) private localStorage: Storage
  ) {

    this.urlServer = 'http://localhost:8080';
    this.token = '';
    this.provider = '';
  }

  header: {};
  getAccessToken(): string | null {
    return this.localStorage.getItem('access_token');
  }

  setAccessToken(token: string | null): void {
    if (token !== null) {
      this.localStorage.setItem('access_token', token);
    } else {
      this.localStorage.removeItem('access_token');
```

```
    }
  }
  clearToken() {
    this.localStorage.removeItem('access_token');

  }


  token: string = '';
  urlServer: string;
  provider: string;



  get(url: string): any {
    return this.http.get<string>(`${this.urlServer}${url}`,
{responseType: 'text' as 'json'})
    .pipe(
      map((response:any)=>
    response as string
    ))
 }
/*se non mappiamo la risposta riceveremo errore 200 json not
valid o un errore simile che causa non pochi problemi dato che è
difficile da andare a rintracciare ed anche da trovare online*/


  //rendiamo la funzione generica cosi da non doverla creare più
volte
  getToken(providerPassed:string, codeValue: string): any {
    return this.http
      .get(`${this.urlServer}/${providerPassed}/getToken`, {
        params: { codeValue },
        observe: 'response',
      })
      .pipe(
        map((response: HttpResponse<any>) => {
          if (response.status == 200 && response.body !== null) {
            this.token = response.body.token;
            this.setAccessToken(this.token);
            this.provider=providerPassed;
            return true;
          } else {
```

```
            return false;
          }
        })
      );
  }


  getProvider():any
  {
    return this.provider;
  }

  /*questa è la funzione dove andremo a richiamare gli endpoint
dove otteniamo le informazioni degli utenti, sfruttando il
provider possiamo renderla generica cosi da richiamarla solo una
volta per tutti*/
  getUserInfo(): Observable<any> {
    const providerToUser = this.provider
    const token = this.getAccessToken();
    if (!token) {
      throw new Error('No access token found');
    }

    return
this.http.get(`${this.urlServer}/${providerToUser}/userInfo`, {
params: { accessToken: token } });

  }
```

ora ci servirà gestire authGuard per poter accedere alle pagine private e dato che dobbiamo verificare i token dei nostri provider per avere un codice più leggibile ci conviene creare un nuovo Service che si occuperà solamente di riconoscere del provider alla quale appartiene il token e poi richiamare l'endpoint nel quale verrà verificato, anche in questo caso cercheremo di creare una funzione generica cosi da non dover creare funzioni per ogni verifica, ed inoltre metteremo da subito il controllo del provider di appartenenza del token

```
import { HttpClient, HttpParams } from '@angular/common/http';
import {  Injectable } from '@angular/core';
import { MyHttpClientService } from './MyHttpClientService';
@Injectable({
  providedIn: 'root',
```

```
})
export class AuthService {

  private googleTokenInfoUrl =
'https://oauth2.googleapis.com/tokeninfo';

 private backendVerifyFacebookTokenUrl =
'http://localhost:8080/facebook/verifyToken'
 private backendVerifyMicrosoftTokenUrl =
'http://localhost:8080/azure/verifyToken'

  constructor(private http: HttpClient,private service:
MyHttpClientService,
  ) {}

verifyToken(){
     const provider = this.service.getProvider()
    let urlProvider =
        provider==='google'?this.googleTokenInfoUrl:
        provider==='azure'?this.backendVerifyMicrosoftTokenUrl:
        this.backendVerifyFacebookTokenUrl
    return this.http.get(`${urlProvider}`,
{'params':{'access_token':this.getAuthToken()}})

}
  getAuthToken(): any {
    return this.service.getAccessToken();
  }
}
```

ora andiamo ad impostare la pagina di home/login,
per prima cosa andiamo nel componente dove creiamo un json che gestirà i provider ed i relativi url, una funzione per andare a richiamare gli endpoint, e due funzioni facoltative che serviranno per navigare nella pagina privata nel caso fossimo tornati nella home senza fare il logout ed un'altra che ci permetterà di eseguire il logout dalla home. implementeremo onInit di angular per far si che che appena entreremo nella pagina verrà richiamata la funzione per ottenere gli url, altra cosa molto importante, per navigare tra le pagine attenti a non importare router da express ma da @angular/router altrimenti riceverete errore visto che non è la funzione che serve a noi per navigare tra le pagine

```
import { Router } from '@angular/router';
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
```

```
  styleUrl: './home.component.scss'
})
export class HomeComponent implements OnInit {
  url: any;
  constructor(private http: MyHttpClientService, private route:
Router) {
    this.url = {'google':'',
                'facebook':'',
                'azure':'',
    };
  }
  ngOnInit(): void {
    this.getUrl();
  }
 getUrl() {
    this.http.get('/google/url').subscribe((data: any) => {
      this.url['google'] = data.url;
});
//inseriamo già gli endpoint ed ogni volta che aggiungiamo il
controller apposito usciamo il richiamo della funzione dal
commento
   /* this.http.get('/azure/url').subscribe((data: any) => {
      console.log(data);
      this.url['azure'] = data;
    });
          this.http.get('/facebook/url').subscribe((data: any) =>
{
      this.url['facebook'] = data;*/

  logout() {
    this.http.clearToken();
  }


  navigateToPrivate() {
    this.route.navigate(['/private'], { replaceUrl: true });
  }
}
}
```

e impostiamo ora il lato html

```
<div class="flex w-8 justify-content-between align-content-center
bg-cyan-700">
```

```
    <div class=" flex flex-column w-7 ">
      <div class=" align-content-center justify-content-center">
      <div class="flex justify-content-center border-top-2 p-3">
        <div>
          <h4>login with socials</h4>

          <div>
            <button pButton>
              <a href="{{ url['google'] }}">
                <i class="pi pi-google"> GOOGLE</i>
              </a>
            </button>

            <button pButton>
              <a href="{{ url['facebook'] }}">
                <i class="pi pi-facebook"></i> FACEBOOK</a
              >
            </button>
            <button pButton>
              <a href="{{ url['azure'] }}">
                <i class="pi pi-microsoft"></i> MICROSOFT</a
              >
            </button>
          </div>
        </div>
      </div>
    </div>
  </div>
    <div >
      <button pButton (click)="navigateToPrivate()"
label="private"></button>
      <button pButton (click)="logout()" label="logout"></button>
    </div>
  </div>
```

ora se proverete ad avviare il programma quasi sicuro non funzionerà anche se avete configurato il cors per permettere l'accesso tramite il localhost del vostro frontend, quindi vi servirà creare un file chiamato proxy.conf.json all'altezza dei file json di configurazione dell'app frontend, il quale conterrà il seguente pezzo di codice

```
{
    "/":
```

```
    {
        "target":"http:localhost:8080",
        "secure": false
    }
}
```

e poi andare a scrivere nello script di avvio, quindi dentro il file package.json nello script “start”, cambieremo il valore che ha con il seguente script

```
"ng serve && --proxy-config proxy.conf.json --host=127.0.0.1"
```

ora andremo a verificare che il link di google funzioni correttamente, quindi se riuscirete a loggare nella pagina di loading sarà arrivato il momento di poter salvare il token di accesso per poi entrare nella pagina privata e ricevere i dati utente:
Se non riuscite ad accedere controllate bene gli errori, io nel mio caso ho dovuto mettere dei console.log e system.out nella api a causa di un errore che andavo a ricevere per il quale ho cambiato la funzione get in quella finale che state vedendo sopra

una volta che Verificate che venite reindirizzate alla pagina di loading passiamo all’aggiungere il richiamo dell’endpoint per ottenere il token di accesso, per prima cosa verificheremo da quale provider stiamo ricevendo la richiesta, il riconoscimento avviene tramite i params che ci arrivano nell’accesso della pagina, ogni provider li avrà configurati in modo differente quindi sono facilmente riconoscibili,
però prima di mettere l’indirizzamento alla pagina successiva vi consiglio di controllare che la richiesta per l’ottenimento del token vada a buon fine,

```
@Component({
  selector: 'app-loading',
  templateUrl: './loading.component.html',
  styleUrl: './loading.component.scss'
})
export class LoadingComponent implements OnInit{


  constructor(private http: MyHttpClientService, private
route:ActivatedRoute, private navigate:Router)
  {
    this.setToken=''
  }
  setToken: any;
  provider: any
  ngOnInit(): void {

      this.route.queryParams.subscribe((params) => {
        console.log(params)
        if (params['code'] !== undefined && params['scope']) {
          this.provider='google'
```

```
        }
        else  if (params['code'] !== undefined && params['state'])
        {
          this.provider='azure'
        }
        else{
          this.provider='facebook'
          }


this.http.getToken(this.provider,params['code']).subscribe((result:
any) => {
              this.setToken= result
              }
            );
      })

      console.log(this.setToken)/*questo token in realtà ci serve solo
per controllare che sia andato tutto a buon fine*/
    this.navigate.navigate(['/private'])
    }
  }
```

per farlo lasciate commentato il pezzo di che manda alla pagina successiva ovvero **this.navigate.navigate([‘/private’])** e controllate nella console cosa vi arriva, se per caso ricevete l’errore nella chiamata dell’api vi consiglio di commentare quel blocco di codice cosi da testare tramite postman l’endpoint per ottenere il token di accesso manualmente, il token che ricevete nella pagina di loading si può usare solamente una volta quindi se non lo commentate e provate a testare sicuramente riceverete errore perché sarà già stato utilizzato, per ottenere il token di accesso di google l’uri è: https://oauth2.googleapis.com/token,

configurate postman nel seguente modo per testarlo

Se tutto funziona bene e venite reindirizzati alla pagina privata allora possiamo passare all'ottenere le informazioni dell'utente, per come abbiamo visto sopra dato che usiamo un solo tipo di utente finale ed una sola funzione per ottenere le informazioni ci basterà creare questa pagina una volta sola senza dover fare cambiamenti in futuro e ci limiteremo ad usarla per fare i testing finali ogni volta che verifichiamo che i token di accesso vengono ottenuti e sono verificati
quindi il componente della pagina privata sarà con la chiamata all'endpoint una volta creata la pagina e poi con una funzione di logout nel caso si volesse uscire:

```
export class PrivateComponent implements OnInit {
  userInfo: any;
  constructor(private http: MyHttpClientService, private route: Router)
{}
  ngOnInit(): void {
    this.http.getUserInfo().subscribe((data) => {
      this.userInfo = data;
      console.log('User Info:', this.userInfo);
    });
  }

  logout() {

    this.http.clearToken();
    this.route.navigate(['/'], { replaceUrl: true })

  }
```

mentre nel lato html andremo a mostrare le informazioni, lo stile usato è uno base preso da w3school per le card al momento

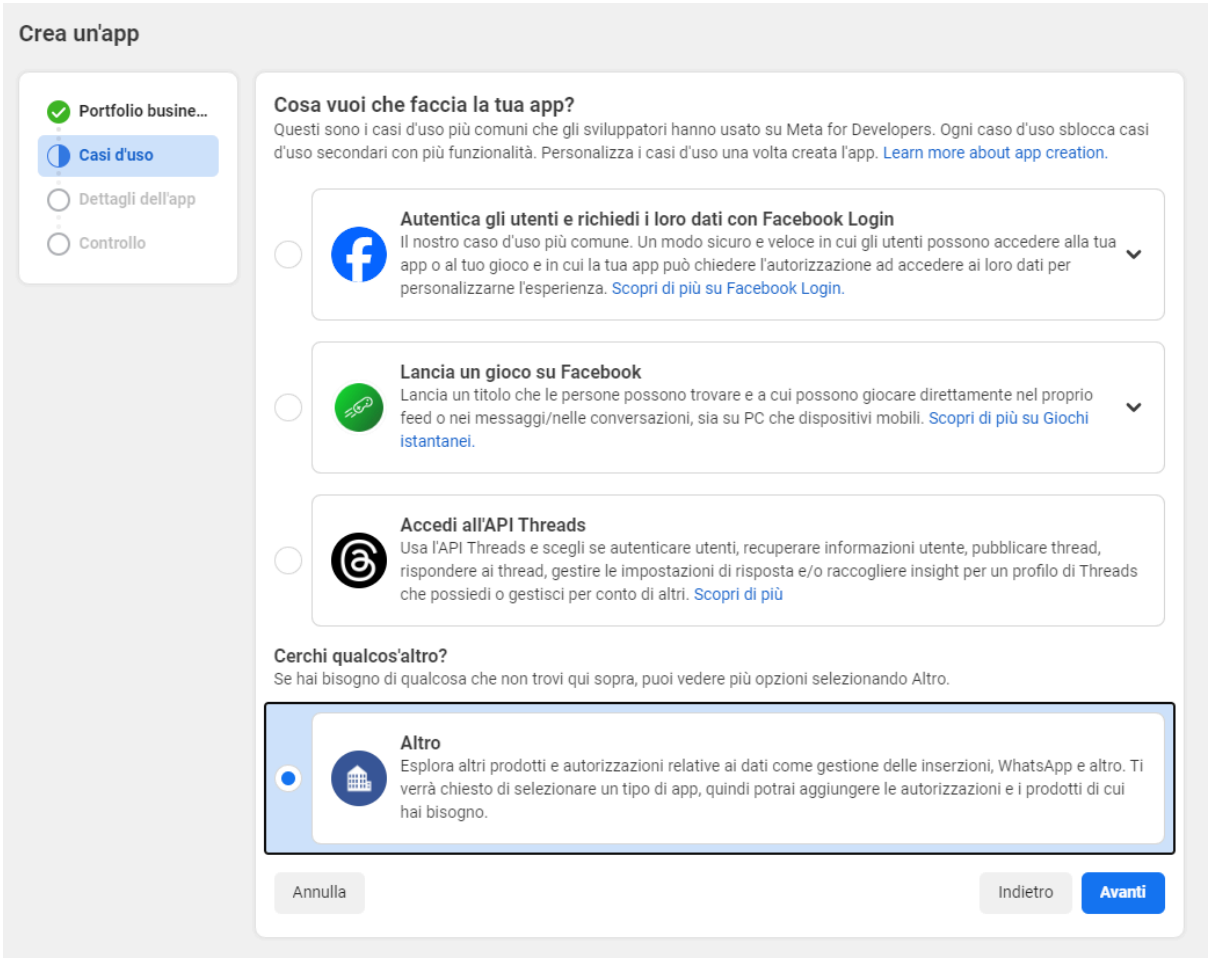
```
  <div  class="card">
    <h2>User Information</h2>
    <img [src]="userInfo.picture" alt="User Picture" />
    <ul>
      <li><strong>Name:</strong> {{ userInfo.name }}</li>
      <li><strong>Surname:</strong> {{ userInfo.surname }}</li>
      <li><strong>Email:</strong> {{ userInfo.email }}</li>
    </ul>
  </div>


<div class="w-full flex justify-content-around">
    <button pButton (click)="logout()">logout</button>
    <button pButton [routerLink]="'/'" label="home without
logout"></button>
  </div>
```

se riuscite a vedere le informazioni del vostro utente google possiamo passare alla configurazione dell'app Facebook

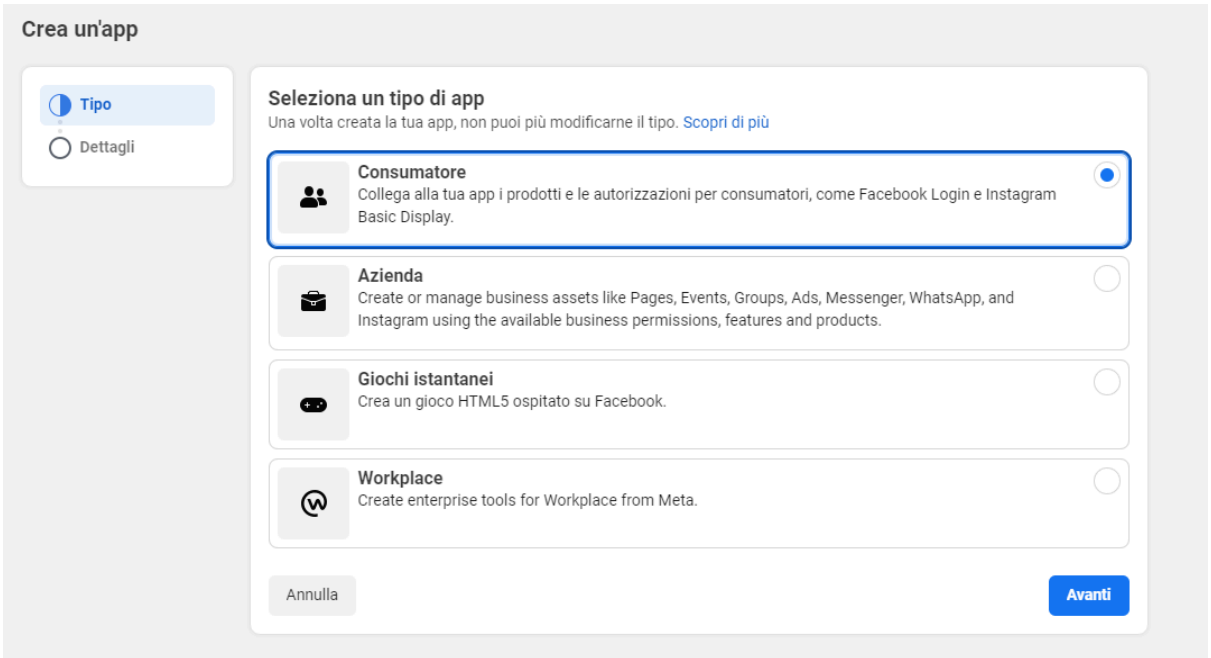
CONFIGURAZIONE PROVIDER DI FACEBOOK

per prima cosa andare nel seguente link per accedere/registrarsi con un account Sviluppatore Meta (totalmente gratuito)
https://developers.facebook.com
Una volta registrati:

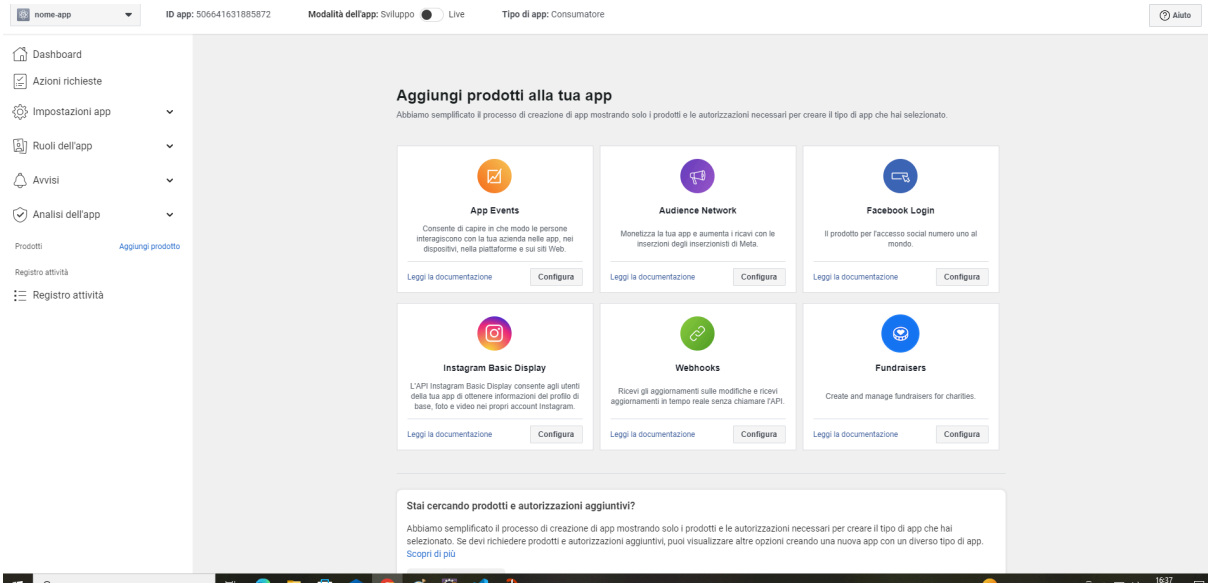
→ Le mie app (in alto a sinistra della schermata da questo link una volta eseguito l'accesso) → crea un'app sempre a destra dello schermo una volta passati nella pagina "Le mie app" → scegliere un portfolio business da collegare all'app (opzionale, io non l'ho messo) → nella schermata vi verrà richiesto di scegliere poi tra le seguenti opzioni:



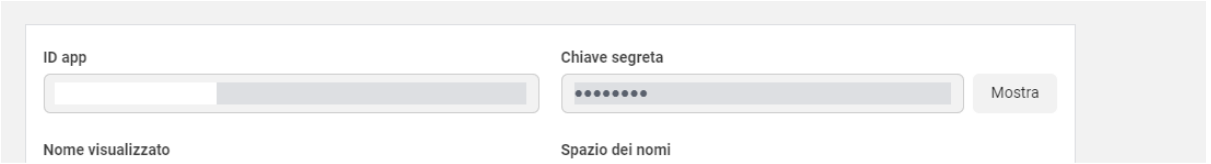
a primo impatto uno potrebbe pensare che sia la prima opzione la migliore per la gestione del Single Sign On tramite i social, ma in realtà conviene scegliere Altro perché la configurazione sarà più intuitiva e veloce da effettuare → selezionare il tipo di app che si vuole creare, in questo caso ci interessa la prima opzione: consumatore



→ inserire il nome che si vuole assegnare all'app e la vostra email di contatto → appena avrete creato cosi la vostra app avrete di fronte la seguente schermata:



quindi selezionare facebook login per andarlo a configurare → vi chiederà per che piattaforma vorrete svilupparla ed in questo caso metteremo web
→ mettete l'uri della vostra app front end (quindi non la pagina di reindirizzamento e poi continuare a cliccare sul tasto per andare avanti finché non arriverete alla schermata "passaggi successivi, a quel punto dovrete solo cliccare nella barra di sinistra su → impostazioni app → base e da li vedrete il vostro client-Id e client-Secret



ora possiamo andare a inserirli nel nostro file application.properties e poi andare a creare e configurare il controller

```
spring.security.oauth2.client.registration.facebook.client-id=Il_vostro_client_id
spring.security.oauth2.client.registration.facebook.client-secret=Il_vostro_client_secret
spring.security.oauth2.client.registration.facebook.scope=email,public_profile
```

per prima cosa ci serve creare un DTO per gli utenti facebook, la creazione sarà leggermente più lunga perché all'ottenimento dei dati riceviamo un json innestato ed alcune informazioni tipo il nome ed il cognome dell'utente sono messe incorporate, quindi per evitare di aggiungere classi di supporto o dipendenze aggiuntive creeremo la classe nel seguente modo cosi che dopo non avremo problemi

```
public class FacebookUserDTO {
  @JsonProperty("id")
  private String id;
  @JsonProperty("name")
  private String name;
  @JsonProperty("email")
  private String email;
  @JsonProperty("picture")
  private Map<String, Object> picture;
```

```java
    // Getters e Setters
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public Map<String, Object> getPicture() {
        return picture;
    }

    public void setPicture(Map<String, Object> picture) {
        this.picture = picture;
    }
    // facebook contiene il nome ed il cognome nella stessa parte quindi per separarli ci servirà
    //questa funzione cosi poi da poterlo salvare nel nostro utente generico

    public String[] getNameParts() {
        if (name == null || name.isEmpty()) {
            return new String[] { "", "" };
        }
        String[] parts = name.split(" ", 2);
        return parts.length == 2 ? parts : new String[] { parts[0], "" };
    }

    //dato che facebook ritorna le immagini di profilo come json annidati siamo costretti a creare una funzione apposita
    //per estrarre l'immagine
    public String getPictureUrl() {
        if (picture != null && picture.containsKey("data")) {
            Map<String, Object> data = (Map<String, Object>) picture.get("data");
```

```
        if (data.containsKey("url")) {
            return (String) data.get("url");
        }
    }
    return null;
  }
}
```

quindi questo sarà il risultato finale
ora possiamo creare il controller dove ci faremo dare, un url per poter accedere, una funzione che ci fornisce il token di accesso una volta effettuato il login dal'uri, una funzione che verifica che il token di accesso sia valido, una funzione per ottenere i dati dell'utente e salvarlo nel DB, dato che al contrario di google sia facebook che Azure non hanno delle api pubbliche per verificare i token di accesso saremo costretti a fare questa funzione extra che prima non abbiamo fatto, ma cercheremo di usare la stessa logica che viene utilizzata dalle api di google cosi da non avere problemi quando richiamiamo la funzione generica dal frontend

```
@RestController
@RequestMapping("/facebook")
public class FacebookController {
        @Autowired
        private UserService userService;

        @Value("${spring.security.oauth2.client.registration.facebook.client-id}")
        private String clientId;
        @Value("${spring.security.oauth2.client.registration.facebook.client-secret}")
        private String clientSecret;
        private static final String REDIRECT_URI = "http://localhost:4200/loading";
        private final WebClient webClient;
        public FacebookController(WebClient.Builder webClientBuilder) {
                this.webClient =
webClientBuilder.baseUrl("https://graph.facebook.com").build();
        }

        @GetMapping("/url")
        public String auth() {
                String url = "https://www.facebook.com/v12.0/dialog/oauth?"
                                + "client_id=" + clientId
                                + "&redirect_uri="+ REDIRECT_URI
                                + "&scope=email";
                return url;
        }

  @GetMapping("/getToken")
```

```
/*come potete osservare la creazione degli uri per poter ottenere i token di accesso sono simili a quello di google, ovvero: url di base del provider che vogliamo usare, il path per indicare che vogliamo i token di accesso, passiamo come parametri il clientId e ClientSecret della nostra app creata prima, l'uri di reindirizzamento, ed il codice di autentificazione che viene passato al frontend una volta effettuato l'accesso*/
public ResponseEntity<String> getToken(@RequestParam("codeValue") String codeValue) {
    try {
        String tokenResponse = webClient.post()
            .uri(uriBuilder -> uriBuilder.path("/v12.0/oauth/access_token")
                .queryParam("client_id", clientId)
                .queryParam("client_secret", clientSecret)
                .queryParam("redirect_uri", REDIRECT_URI)
                .queryParam("code", codeValue)
                .build())
            .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_FORM_URLENCODED_VALUE)
            .retrieve()
            .bodyToMono(String.class)
            .block();
        return ResponseEntity.ok(tokenResponse);
    } catch (Exception e) {
        System.out.println("inside the error: " + e.getMessage());
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(e.getMessage());
    }
  }
  @GetMapping("/userInfo")
  public ResponseEntity<UserDTO> getUserInfo(@RequestParam("accessToken") String accessToken) {
    try {
        UserDTO user = findUserInDB(accessToken);
        return ResponseEntity.ok(user);
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
    }
  }

//questa logica che usiamo per verificare il token la riutilizzeremo anche per il provider dopo in maniera quasi identica, ci facciamo tornare un json cosi da renderlo simile al tipo di ritorno di verifyToken di google cosi da non dover creare una funzione in più nel frontend
        @GetMapping("/verifyToken")
```

```
        public ResponseEntity<Map<String, Object>>
verifyToken(@RequestParam("access_token") String token) {
                try {
                        boolean isValid = checkFacebookTokenValidity(token);
                         Map<String, Object> response = new HashMap<>();
                        System.out.print("controlliamo isValid:\n"+isValid);
                        if (isValid) {

                           response.put("valid", isValid);
                                return ResponseEntity.ok(response);
                        } else {
                                 response.put("valid", isValid);
                                return
ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(response);
                        }
                } catch (IOException e) {
                        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
                }
        }
/*come avrete notato stiamo in realtà una chiamata molto simile a quella del
getToken, ma non fatevi ingannare perché per quanto simile stiamo richiamando
un'altra API, che richiede sia il nostro clientId che il clientSecret, motivo per il quale
non possiamo richiamarla nel frontend*/
        private boolean checkFacebookTokenValidity(String token) throws
IOException {
                 String tokenInfoUrl =
String.format("/v12.0/debug_token?input_token=%s&access_token=%s|%s", token,
clientId, clientSecret);
                System.out.print("url:\n"+tokenInfoUrl+"\n\n");
                String tokenResponse = webClient.get()
                        .uri(tokenInfoUrl)
                        .retrieve()
                        .bodyToMono(String.class)
                        .block();
                ObjectMapper objectMapper = new ObjectMapper();
                JsonNode jsonNode = objectMapper.readTree(tokenResponse);
                JsonNode dataNode = jsonNode.path("data");
                System.out.print("token Response:\n"+tokenResponse+"\n");
                return dataNode.path("is_valid").asBoolean(false);
            }

//il
  private UserDTO findUserInDB(String accessToken) throws Exception {
```

```
/*qui stiamo andando a mappare l'utente come abbiamo fatto con l'utente google, quindi ricordatevi che appena avrete scritto questa parte dovrete andare ad aggiungere nell'userDTO un costruttore dove convertite l'utente Facebook in utente normale se non l'avete già fatto*/
UserDTO userInfo = new UserDTO(webClient.get()
        .uri(uriBuilder -> uriBuilder.path("/me")
            .queryParam("fields", "id,name,email")
            .queryParam("access_token", accessToken)
            .build())
        .retrieve()
        .bodyToMono(FacebookUserDTO.class)
        .block());
    System.out.println("email: " + userInfo.getEmail());
    userInfo.setProvider("facebook");
    Optional<User> user = userService.findByEmail(userInfo.getEmail());
    if (user.isEmpty()) {
        String response = userService.registerAccount(userInfo);
    }
    return userInfo;
  }
}
```

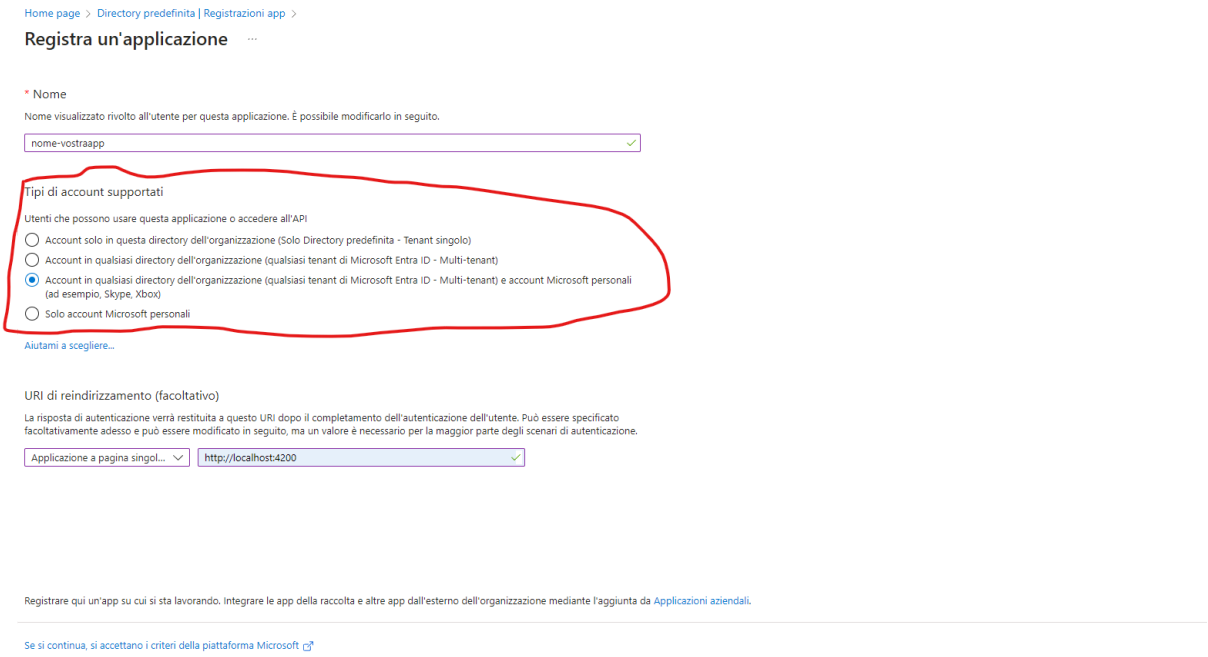
al solito come per google vi consiglio di provare per step, ovvero prima controllate che l'url vi reindirizzi nel posto gioco e vi consegni il token di autentificazione, dopo verificate che vi venga dato il token di accesso, poi verificate che il token di accesso venga confermato valido e solo a quel punto cercate di vedere se gli utenti vengono ritornati.
ora passiamo alla configurazione più ostica quella di Azure

CONFIGURAZIONE PROVIDER AZURE

azure richiederà parecchie cose in più anche solo nella configurazione dell'url e nella creazione dell'app,

per prima cosa andate nel seguente link:
www.portal.azure.com

dovrete creare un profilo sviluppatore azure, potete farlo gratuitamente se siete studenti, una volta che avete eseguito l'accesso e sarete nella home dovrete in ordine:
cercare **Microsoft Entra Id** → Una volta trovato, entrateci e nella barra di sinistra cercate **Registrazioni App** → nuova registrazione e qui molto importante oltre al nome della vostra app,vi verranno richiesti i tipi di account supportati, dovrete selezionare di qualsiasi Directory e Account privati



per l'uri di reindirizzamento invece segnate Applicazione a pagina singola e segnate la pagina di reindirizzamento che usate, nel nostro caso è http://localhost:4200/loading → una volta creata vedrete nella schermata il vostro **idTenat** e **ClientId**, potete già salvarli in un file di testo ma prima di andarli ad utilizzare dovrete su **Autenticazione** → cercate **flussi di accesso** ed impostateli su Token ID (flussi implicit ed ibridi), una volta salvato → andate a verificare su **Autorizzazioni API** di avere il permesso User.Read per poter leggere le informazioni dell'utente, in caso contrario dovrete aggiungerlo da aggiungi un'autorizzazione → autorizzazioni delegate → User
ma normalmente lo mette di default se create l'app seguendo i passaggi scritti qui
a questo punto andate su **Certificati e Segreti** → nuovo segreto client, assegnate il nome che preferite, in questo caso l'ho chiamato client secret, ed appena lo generate salvate subito il valore che vedete, ritornare un secondo momento in questa schermata non vi permetterà di salvare il valore e sarete costretti e ricrearlo da 0.

Ora che abbiamo finito di creare la nostra app possiamo iniziare a configurarla nel backend, quindi per prima cosa nel file application.properties ci servirà aggiungere le seguenti cose:

```
azure.ad.client-id=Il_Vostro_Client_tID
azure.ad.client-secret=il_Vostro_Client_Secret
azure.ad.redirect-uri=http://localhost:4200/loading (opzionale)
azure.ad.tenant-id=il_Vostro_Tenant_Id

azure.ad.token-endpoint=https://login.microsoftonline.com/${azure.ad.tenant-id}/oauth2/v2.0/token
azure.ad.userinfo-endpoint=https://graph.microsoft.com/v1.0/me
```

ora creiamo il DTO di supporto che ci servirà per Azure visto che quando richiediamo un utente ci vengono fornite queste informazioni:

```
@JsonIgnoreProperties(ignoreUnknown = true)
```

```java
public class AzureUserDTO {
    @JsonProperty("sub")
    private String sub;
    @JsonProperty("@odata.context")
    private String odataContext;
    @JsonProperty("givenname")
    private String givenName;
    @JsonProperty("familyname")
    private String familyName;
    @JsonProperty("email")
    private String email;
    @JsonProperty("locale")
    private String locale;
    @JsonProperty("picture")
    private String picture;
    @JsonProperty("name")
    private String name;
    @JsonProperty("email_verified")
    private boolean emailVerified;

    public AzureUserDTO() {}

    public String getSub() {
        return sub;
    }
    public void setSub(String sub) {
        this.sub = sub;
    }
    public String getOdataContext() {
        return odataContext;
    }
    public void setOdataContext(String odataContext) {
        this.odataContext = odataContext;
    }
    public String getGivenName() {
        return givenName;
    }
    public void setGivenName(String givenName) {
        this.givenName = givenName;
    }
    public String getFamilyName() {
        return familyName;
    }
    public void setFamilyName(String familyName) {
```

```
    this.familyName = familyName;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public String getLocale() {
    return locale;
}
public void setLocale(String locale) {
    this.locale = locale;
}
public String getPicture() {
    return picture;
}
public void setPicture(String picture) {
    this.picture = picture;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public boolean isEmailVerified() {
    return emailVerified;
}
public void setEmailVerified(boolean emailVerified) {
    this.emailVerified = emailVerified;
}
@Override
public String toString() {
    return "AzureUserDto{" +
            "sub='" + sub + '\'' +
            ", odataContext='" + odataContext + '\'' +
            ", givenName='" + givenName + '\'' +
            ", familyName='" + familyName + '\'' +
            ", email='" + email + '\'' +
            ", locale='" + locale + '\'' +
            ", picture='" + picture + '\'' +
            ", name='" + name + '\'' +
            ", emailVerified=" + emailVerified +
```

```
            '}';
    }
```

ed ora possiamo passare nella configurazione del controller, come ho annunciato prima qui ci saranno altre funzioni in più che prima non ci servivano, questo perché quando generiamo l'uri per accedere sono obbligatori da passare un codice di sicurezza e con che tipo di sicurezza è stato criptato, inoltre possiamo anche generare lo stato (opzionale ma l'abbiamo fatto)

```
@RestController
@RequestMapping("/azure")
public class AzureController {
   @Value("${azure.ad.client-id}")
   private String clientId;
   @Value("${azure.ad.client-secret}")
   private String clientSecret;
   @Value("${azure.ad.redirect-uri}")
   private String redirectUri;
   @Value("${azure.ad.token-endpoint}")
   private String tokenEndpoint;
   @Value("${azure.ad.userinfo-endpoint}")
   private String userinfoEndpoint;
   @Autowired
   private UserService userService;
   private final WebClient webClient;
   private final ObjectMapper objectMapper = new ObjectMapper();
   private String []  store = new String [2];
   public AzureController(WebClient.Builder webClientBuilder) {
      this.webClient = webClientBuilder.build();
   }
   @GetMapping("/url")
   public String auth() {
         String codeVerifier = generateCodeVerifier();
            String codeChallenge = generateCodeChallenge(codeVerifier);
            String state = generateState();
            this.store[0] = codeVerifier;//qui salveremo il codice criptato che ci serve
anche dopo per poter ottenere il token di accesso
            this.store[1] = state;

            String
url=UriComponentsBuilder.fromHttpUrl("https://login.microsoftonline.com/consumers/
oauth2/v2.0/authorize")
            .queryParam("client_id", clientId)
            .queryParam("response_type", "code")
            .queryParam("redirect_uri", redirectUri)
```

```
            .queryParam("response_mode", "query")
            .queryParam("scope", "openid profile email")
            .queryParam("state", state)
            .queryParam("code_challenge", codeChallenge) //il codice criptato che
dobbiamo generare, salvare e passare,
            .queryParam("code_challenge_method", "S256")// il modo con la quale
l'abbiamo criptato
            .build()
            .toUriString();
            return  url;
    }
    @GetMapping("/getToken")
    public ResponseEntity<String> getToken(@RequestParam("codeValue") String
codeValue) {
        try {
            System.out.print("codeValue: \n" + codeValue+"\n");

            String codeVerifier = this.store[0];//qua prendiamo il codice che abbiamo
salvato prima
            if (codeVerifier == null) {
                return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Invalid
state.");
            }
//Se non avete seguito tutti i passaggi nella creazione questo punto potrebbe non
funzionarvi
            String tokenResponse = webClient.post()
                    .uri("https://login.microsoftonline.com/consumers/oauth2/v2.0/token")
                    .header(HttpHeaders.CONTENT_TYPE,
MediaType.APPLICATION_FORM_URLENCODED_VALUE)
                    .header(HttpHeaders.ORIGIN, "http://localhost:4200")
                    .bodyValue("code=" + codeValue +
                            "&client_id=" + clientId +
                            "&redirect_uri=" + redirectUri +
                            "&grant_type=authorization_code" +
                            "&code_verifier=" + codeVerifier)
                    .retrieve()
                    .bodyToMono(String.class)
                    .block();
                      JsonNode jsonNode = objectMapper.readTree(tokenResponse);
                      String accessToken = jsonNode.get("access_token").asText();
            System.out.println("Token Response: " + accessToken);
            return ResponseEntity.ok(tokenResponse);
        } catch (WebClientResponseException e) {
```

```
            System.err.println("Error Response Body di getToken: " +
e.getResponseBodyAsString());
            return
ResponseEntity.status(e.getStatusCode()).body(e.getResponseBodyAsString());
        } catch (Exception e) {
            e.printStackTrace();
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(e.getMessa
ge());
        }
    }
    //la funzione di verifica del token  ha lo stesso meccanismo di quella di facebook
come avevo accennato in precedenza
    @GetMapping("/verifyToken")
public ResponseEntity<Map<String, Object>>
verifyToken(@RequestParam("access_token") String token) {
    try {
        boolean isValid = checkAzureTokenValidity(token);
        Map<String, Object> response = new HashMap<>();
        System.out.print("controlliamo isValid:\n" + isValid + "\nsiamo dopo");
        response.put("valid", isValid);
        if (isValid) {
            return ResponseEntity.ok(response);
        } else {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(response);
        }
    } catch (IOException e) {
        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}
private boolean checkAzureTokenValidity(String token) throws IOException {
    String tokenInfoUrl = "https://graph.microsoft.com/oidc/userinfo";
    System.out.print("url:\n" + tokenInfoUrl + "\n\n");
    String tokenResponse = webClient.get()
            .uri(tokenInfoUrl)
            .header(HttpHeaders.AUTHORIZATION, "Bearer " + token)
            .retrieve()
            .bodyToMono(String.class)
            .block();
    ObjectMapper objectMapper = new ObjectMapper();
    JsonNode jsonNode = objectMapper.readTree(tokenResponse);
    // Se non ci sono errori nella risposta, il token è valido
    return jsonNode.has("sub");
```

```
}
//anche l'ottenimento dei dato dell'utente non varia dagli altri provider
   @GetMapping("/userInfo")
   public ResponseEntity<UserDTO> userInfo(@RequestParam("accessToken")
String accessToken) {
      try {
         UserDTO user = findUserInDB(accessToken);
         return ResponseEntity.ok(user);
      } catch (Exception e) {
         return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
      }
   }

//qui sotto ci sono le funzioni che sono necessarie per generare State, Codice di
verificare e Codice criptato
   private String generateState() {
      byte[] randomBytes = new byte[16];
      new SecureRandom().nextBytes(randomBytes);
      return Base64.getUrlEncoder().withoutPadding().encodeToString(randomBytes);
   }
   private String generateCodeVerifier() {
      byte[] randomBytes = new byte[32];
      new SecureRandom().nextBytes(randomBytes);
      return Base64.getUrlEncoder().withoutPadding().encodeToString(randomBytes);
   }
   private String generateCodeChallenge(String codeVerifier) {
      try {
         MessageDigest digest = MessageDigest.getInstance("SHA-256");
         byte[] hashedBytes =
digest.digest(codeVerifier.getBytes(StandardCharsets.US_ASCII));
         return
Base64.getUrlEncoder().withoutPadding().encodeToString(hashedBytes);
      } catch (NoSuchAlgorithmException e) {
         throw new RuntimeException(e);
      }
   }
   private UserDTO findUserInDB(String accessToken) throws Exception {
      UserDTO userInfo = new UserDTO(webClient.get()
            .uri("https://graph.microsoft.com/oidc/userinfo")
            .header(HttpHeaders.AUTHORIZATION, "Bearer " + accessToken)
            .retrieve()
            .bodyToMono(AzureUserDTO.class)
            .block());
      userInfo.setProvider("microsoft");
```

```
    Optional<User> user = userService.findByEmail(userInfo.getEmail());
    if (user.isEmpty()) {
        userService.registerAccount(userInfo);
    }
    return userInfo;
}
```