

# Progetto di Sicurezza delle Architetture Orientate ai Servizi

---

Sofia Gavanelli, Alessio Riccardi

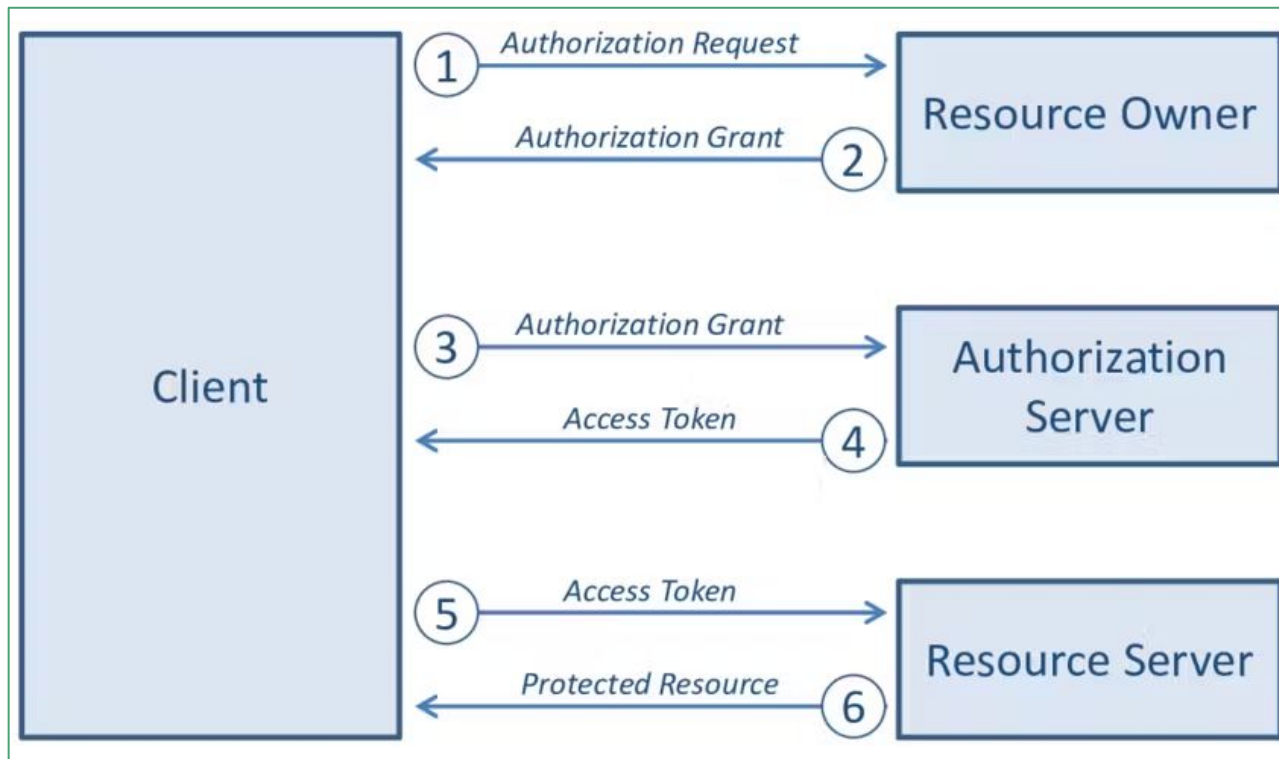
# Obiettivo

L'obiettivo è quello di implementare un progetto che preveda lo sviluppo del protocollo OAuth2.0 per l'autorizzazione e l'accesso alle risorse protette dei servizi web.

In particolare, all'interno del progetto sono stati sviluppati da zero, tutti i componenti principali da cui è composto il protocollo.

# OAuth 2.0

Il framework di autenticazione OAuth 2.0 è un open standard che consente ad un utente (Resource Owner) di permettere ad un sito Web o ad un'applicazione di terze parti (Client) l'accesso alle proprie risorse protette (contenute nel Resource Server), senza necessariamente rivelare a terzi le proprie credenziali o l'identità.



# Sviluppo

All'interno di questo progetto sono stati sviluppati Authorization Server, Resource Server e Security Client grazie all'utilizzo di:

- **Spring**: framework open source per lo sviluppo di applicazioni su piattaforma Java. In particolare, Spring-Boot che è uno strumento che semplifica e velocizza lo sviluppo di applicazioni web e microservizi con Spring Framework tramite *(a) configurazione automatica, (b) un approccio categorico alla configurazione e (c) la capacità di creare applicazioni autonome.*
- **Maven**: strumento di gestione di progetti software basati su Java e build automation. In particolare, si basa sul concetto di “project object model” (POM).
- **MySQL Workbench**: strumento visuale di progettazione per database (integra sviluppo SQL, gestione, modellazione dati, creazione e manutenzione di database MySQL all'interno di un unico ambiente)

# Struttura

Il progetto è composto dai tre moduli, ognuno di esso possiede il suo file `application.yml` che contiene i dati di configurazione di quello specifico modulo.

All'interno sono contenuti la configurazione del database e, nel caso del client, le informazioni della configurazione di sicurezza di oauth:

```
security:
  oauth2:
    client:
      registration:
        api-client-oidc: [...] *
        api-client-authorization-code: [...]
      provider:
        spring:
          issuer-uri: http://auth-server:9000
```

dove openID connect\* è uno strato di autenticazione del framework e insieme all'authorization-code costituisce il lato di registrazione del client

# Auth-server

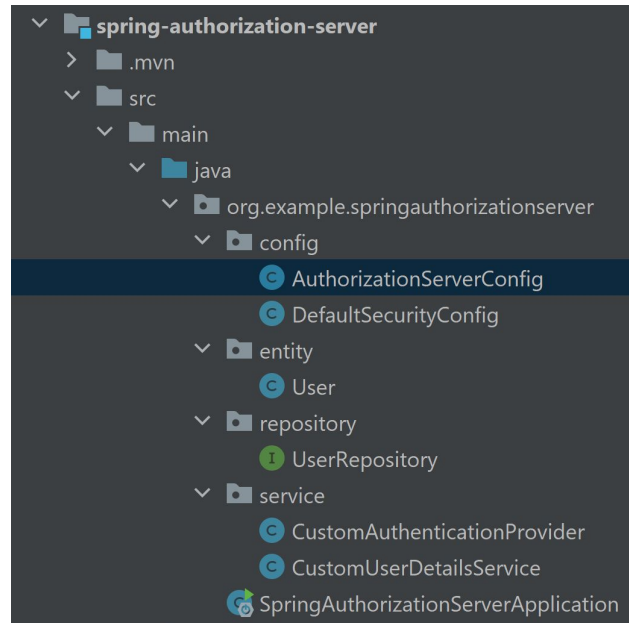
L'Authorization Server gira su auth-server:9000 ed è composto dai package visibili in foto.

In particolare nell'AuthorizationServerConfig viene gestita la configurazione standard per la generazione di chiave pubblica e privata tramite:

```
RSAPublicKey rsaKey = generateRsa();  
JWKSet jwkSet = new JWKSet(rsaKey);
```

DefaultSecurityConfig contiene invece la configurazione di base dell'authorization server.

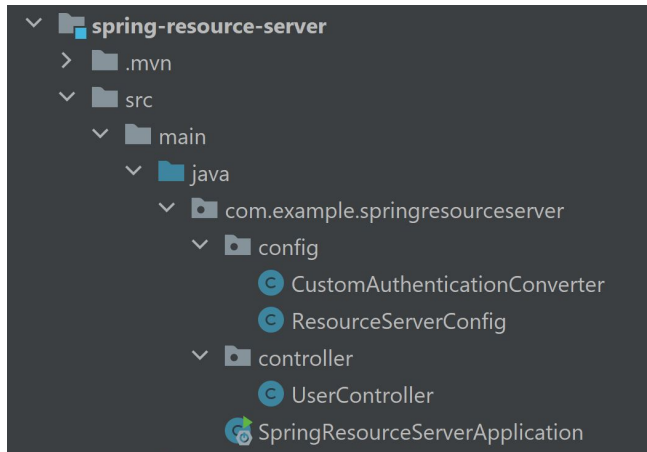
UserRepository è la classe necessaria per la gestione dell'utente nel database.



# Resource-server

Le “risorse”, nel caso di questo progetto, sono semplicemente pagine diverse nel caso di ruoli diversi:

- /users nel caso di utente semplice che restituisce l'elenco degli utenti
- /managers nel caso di manager che restituisce semplicemente un utente in più rispetto agli utenti ottenuti in /users



# Security-client

Infine c'è la registrazione del client gestita dal security-client. Questo modulo si occupa di fornire la registrazione dell'utente tramite la classe RegistrationController e fornisce il mapping delle varie pagine del servizio tramite classe HelloController.

Il client è registrato nell'Authorization Server tramite il Bean `RegisteredClientRepository`. In quest'ultimo, vengono specificate alcune informazioni, come nome client, gli scopes, gli Authorization Grant e l'uri a cui si viene reindirizzati dopo l'autenticazione.

```
//Configurazione base per la registrazione del client
@Bean
public RegisteredClientRepository registeredClientRepository() {
    RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("api-client") //nome client
        .clientSecret(passwordEncoder.encode( rawPassword: "secret")) //client password
        .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
        .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
        .authorizationGrantType(AuthorizationGrantType.PASSWORD)
        .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
        .redirectUri("http://127.0.0.1:8080/login/oauth2/code/api-client-oidc")
        .redirectUri("http://127.0.0.1:8080/authorized")
        .scope(OpenidScopes.OPENID)
        .scope("api.read")
        .clientSettings(ClientSettings.builder().requireAuthorizationConsent(true).build())
        .build();

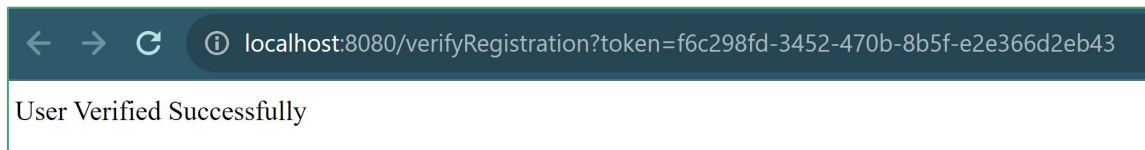
    //In memory dell'authorization server
    return new InMemoryRegisteredClientRepository(registeredClient);
}
```



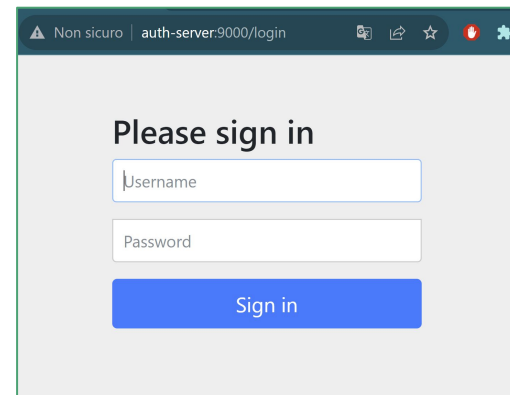
# Funzionamento

Una volta avviato, l'esecuzione del programma avviene nel seguente modo:

- registrazione: **POST** <http://localhost:8080/register> con le informazioni dell'utente - UserModel, ovvero username, password (che viene memorizzata criptata nel DB) e la mail
- verifica dell'utente tramite la chiamata del servizio **/verifyRegistration**



- dopodichè una volta registrato, se l'utente provasse ad accedere ad una delle risorse protette, verrebbe re-indirizzato alla pagina di login per l'inserimento delle credenziali (<http://auth-server:9000/login>)

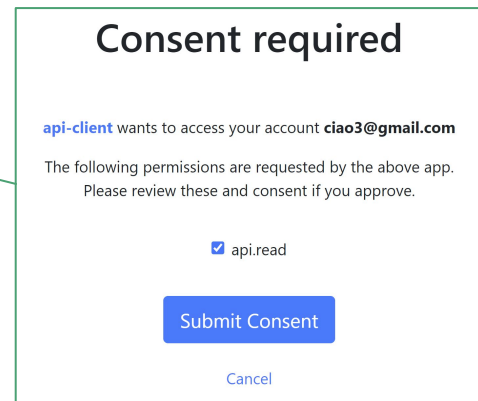


A questo punto inizia il vero protocollo OAuth:

- L'applicazione (Client) richiede l'autorizzazione dell'utente per accedere alle risorse utili al servizio
- Se l'utente autorizza la richiesta, l'applicazione riceve un **authorization grant** (authorization code)
- L'applicazione, presentando l'authorization code unito alla sua identità all'authorization server, richiede un **access token**

=> Se l'identità dell'applicazione è confermata e l'authorization code è valido, l'authorization server invia l'access token: la fase di autorizzazione è conclusa

- L'applicazione richiede la risorsa al resource server presentando l'access token: se l'access token è valido il resource server presenta la *risorsa* richiesta che nel nostro caso è la *pagina /users*



Name	Status	Type	Initiator	Size	Time
hello	302	document / Redirect	Other	598 B	6 ms
authorize?response_type=code&client_id=api-client&...%3D&redire...	302	document / Redirect	hello	534 B	8 ms
authorized?code=0yAH2Rw97Pfw_t5f5mjDW0baZ69_EU8I79...te=V-...	302	document / Redirect	auth-server:9000/oauth2/authoriz...	431 B	558 ms
hello	200	document	authorized?code=0yAH2Rw97Pfw...	470 B	1.22 s

# Autorizzazioni e ruoli(1)

Sono stati introdotti due diversi ruoli: UTENTE e MANAGER per differenziare l'accesso e le risorse che si ha la possibilità di visualizzare.

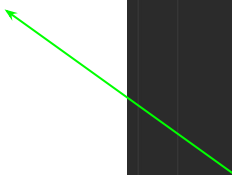
In particolare, l'utente con ruolo UTENTE dovrebbe essere in grado di accedere alle risorse visibili sotto la pagina /users, mentre l'utente con ruolo MANAGER, dovrebbe avere accesso alle risorse visibili alla pagina /managers e /users.

Per garantire questa differenziazione, sono stati provati due metodi:

- il solo utilizzo di antMatchers() con la specifica del metodo hasRole()/hasAnyRole() o l'utilizzo dell'annotazione @Secured(ruolo\_utente)
- la customizzazione del token con l'aggiunta del claim "roles" oltre all'utilizzo come in precedenza degli antMatchers

## Autorizzazioni e ruoli(2)

Come si può vedere in foto, in questo caso è stato utilizzato l'approccio con il metodo `antMatchers()`, in cui è stato specificato per la pagina `/users` il ruolo sia di `MANAGER` che di `USER`, mentre per la pagina `/managers` solo il ruolo di `MANAGER`.



```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .cors() CorsConfigurer<HttpSecurity>
        .and() HttpSecurity
        .csrf() CsrfConfigurer<HttpSecurity>
        .disable() HttpSecurity
        .authorizeHttpRequests() AuthorizeHttpRequestsConfigurer<...>.AuthorizationManagerRequestMatcherRegistry
        .antMatchers(WHITE_LIST_URLS).permitAll()
        .antMatchers(...antPatterns: "/home").permitAll()
        .antMatchers(...antPatterns: "/protected").hasAnyRole(...roles: "MANAGER", "USER")
        .antMatchers(...antPatterns: "/manager").hasAnyRole(...roles: "MANAGER")
        .antMatchers(...antPatterns: "/user").hasRole("USER")
        .antMatchers(...antPatterns: "/managers").hasRole("MANAGER")
        .antMatchers(...antPatterns: "/users").hasAnyRole(...roles: "USER", "MANAGER")
        .and() HttpSecurity
        .formLogin() FormLoginConfigurer<HttpSecurity>
        .and() HttpSecurity
        .oauth2Login(oauth2Login ->
            oauth2login.LoginPage("/oauth2/authorization/api-client-oidc"))
        .oauth2Client(Customizer.withDefaults());

    return http.build();
}
```

## Autorizzazioni e ruoli(3)

```
@Bean
OAuth2TokenCustomizer<JwtEncodingContext> jwtCustomizer() {
    return context -> {
        if (context.getTokenType() == OAuth2TokenType.ACCESS_TOKEN) {
            Authentication principal = context.getPrincipal();
            Set<String> authorities = principal.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority)
                .collect(Collectors.toSet());
            context.getClaims().claim("roles", authorities);
        }
    };
}
```

In questo caso invece, abbiamo provato ad aggiungere il claim “roles” al token, in modo tale da estrarli una volta ottenuto il token di accesso. Anche in questo caso sono stati utilizzati i metodi antMatchers come visto nella slide precedente.

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .mvcMatchers("/users")
        .access("hasAuthority('SCOPE_api.read')")
        .mvcMatchers("/managers")
        .access("hasAuthority('SCOPE_api.read')")
        .and()
        .oauth2ResourceServer()
        .jwt()
        .jwtAuthenticationConverter(new CustomAuthenticationConverter());
    return http.build();
}
```

# Conclusioni

In conclusione abbiamo implementato i componenti principali di cui si compone il protocollo OAuth2.0, ma a causa di alcuni problemi a cui non siamo riusciti a trovare una soluzione che funzionasse correttamente, non siamo stati in grado di testare a pieno la robustezza del protocollo e quindi a fornire autorizzazioni diverse in base al ruolo dell'utente.

GRAZIE PER L'ATTENZIONE