Le Circolari
Progetto SOASec

Simone Galimberti
Stefano Giardina
Mattia Ruo

Indice

Specifiche dell'applicazione

L'obiettivo è stato quello di implementare un'applicazione che utilizzasse il protocollo OAuth2.0 per la gestione dell'autenticazione e l'autorizzazione per l'accesso a risorse protette in base al ruolo dell'utente. La **nostra applicazione** consente di gestire l'upload o la consultazione delle circolari pubblicate agli utenti registrati.
Gli utenti sono suddivisi in 4 ruoli:

- **Amministratore**: registrazione nuovi utenti + visualizzazione elenco completo degli utenti registrati con possibilità di modifica ed eliminazione.
- **Personale Presidenza**: upload di nuove circolari + visualizzazione elenco completo delle circolari caricate con possibilità di modifica ed eliminazione.
- **Studente** e **Docente**: visualizzazione delle circolari di competenza

Metodologie di sviluppo e strumenti utilizzati

- **Spring**: framework open-source, insieme di librerie che forniscono molteplici funzionalità, dal semplificare lo sviluppo di applicazioni web (*Spring-Boot*), all'interazione con il database (*Spring-Data*) fino alla gestione della catena di sicurezza di un 'applicazione (*Spring-Security*).
- **MySQL**: DBMS che si basa sul linguaggio SQL. Nel nostro caso si è scelto di utilizzare l'interfaccia "phpMyAdmin" che consente di amministrare e progettare i database in maniera semplice.
- **REST API**: interfacce che consentono di esporre le risorse in maniera sicura. Si basano sul protocollo HTTP e le risorse sono mappate tramite endpoint (es: /studente/api/circolari). Le operazioni eseguibili sulle risorse sono descritte dai metodi HTTP: GET, POST, PUT e DELETE.

Spring Boot

Java Spring Framework è un framework open source di livello aziendale molto diffuso per la creazione di applicazioni autonome. **Spring Boot** è uno strumento che semplifica e velocizza lo sviluppo di **applicazioni web** e **microservizi** tramite tre funzionalità principali:

- Configurazione automatica
- Un approccio categorico alla configurazione
- La capacità di creare applicazioni autonome

Queste funzionalità cooperano per fornire uno strumento che consente di configurare applicazioni basate su Spring che richiedono una configurazione e un'installazione minime.

Spring Security

Spring Security è un potente framework di sicurezza per applicazioni Java che fornisce **autenticazione**, **autorizzazione** e protezione dai principali attacchi informatici.
È un componente essenziale del vasto ecosistema Spring e si integra perfettamente con Spring Boot.
Il framework fornisce infatti meccanismi di autenticazione basati su username e password, OAuth2, SAML e molto altro.
Una volta che l'utente si è autenticato, Spring Security gestisce l'**autorizzazione** tramite il concetto di "**ruoli**" e "**permessi**". È possibile definire chiaramente quali utenti possono accedere a determinate risorse e funzionalità dell'applicazione, garantendo così la tutela dei dati sensibili e la corretta separazione dei compiti.

OAuth2.0

Il framework di autenticazione OAuth 2.0 è uno standard aperto che consente ad un utente di concedere ad un sito Web o ad un'applicazione di terze parti l'accesso alle proprie risorse protette, senza necessariamente rivelare a terzi le proprie credenziali. È un protocollo di autenticazione basato sui token. Il token è una stringa, firmata da un server per le verifiche di integrità, che può contenere molte informazioni sull'utente, quali le autorizzazioni.

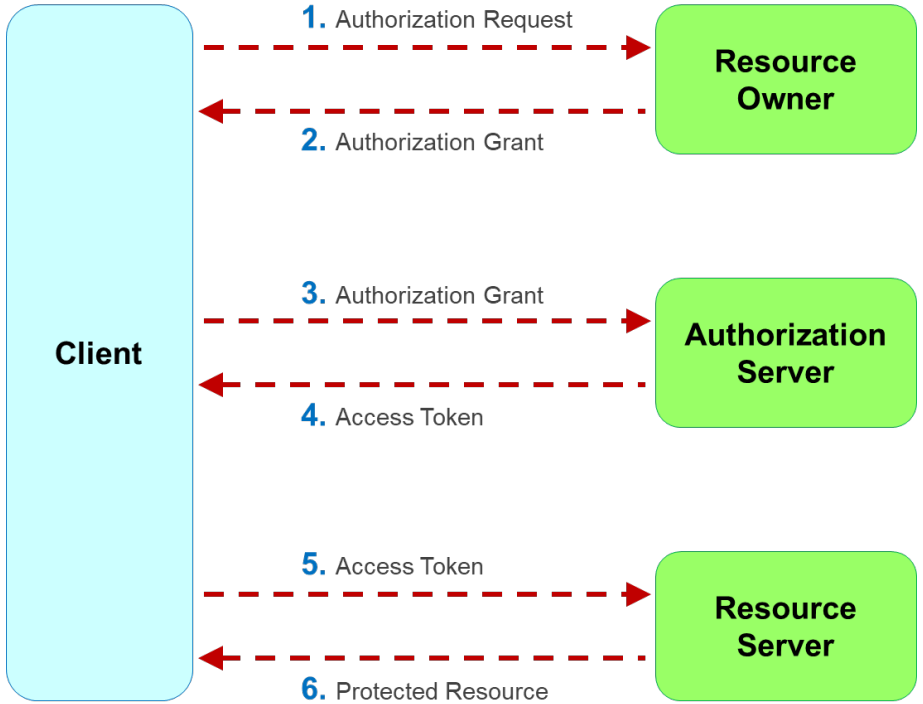


Figura: Flusso protocollo OAuth2.0

I componenti dell'applicazione

All'interno del progetto sono stati sviluppati da zero tutti i componenti principali da cui è composto il protocollo OAuth2.0:

- **Authorization Server**
- **Resource Server**
- **Client**

AUTHORIZATION SERVER

Introduzione

Nel protocollo OAuth 2.0, il server di autorizzazione (Authorization Server) assume un ruolo cruciale nel processo di autenticazione e autorizzazione. Funge da intermediario sicuro tra l'utente, l'applicazione client e il server di risorse, garantendo un accesso controllato e affidabile alle informazioni protette.
Nel nostro caso, il server è in ascolto sulla porta 9000.

Autenticazione OAuth - Configurazione

Questo codice definisce un bean Spring che configura la sicurezza dell'applicazione. Protegge le risorse dell'applicazione utilizzando l'autenticazione OAuth 2.0 e l'autorizzazione basata su token JWT, fornendo un livello di sicurezza completo.

```
@Bean
@Order(1)
public SecurityFilterChain asSecurityFilterChain(HttpSecurity http) throws Exception {

    OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);

    return http
            .getConfigurer(OAuth2AuthorizationServerConfigurer.class).oidc(withDefaults())
            .and()
            .exceptionHandling(e -> e
                    .authenticationEntryPoint(new LoginUrlAuthenticationEntryPoint("/login")))
            .oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt)
            .build();
}
```

Autenticazione OAuth - Login

Questo codice abilita l'autenticazione tramite form login e stabilisce che tutte le richieste debbano essere effettuate da utenti autenticati. Fornisce un livello di sicurezza aggiuntivo rispetto alprecedente bean, basato sull'autenticazione standard tramite form login.

```
@Bean
@Order(2)
public SecurityFilterChain appSecurityFilterChain(HttpSecurity http) throws Exception {
    return http
            .formLogin(withDefaults())
            .authorizeHttpRequests(authorize ->authorize.anyRequest().authenticated())
            .build();

}
```

Autenticazione OAuth - Registrazione client OAuth2.0

d Il metodo registeredClientRepository definisce un RegisteredClientRepository che memorizza i dettagli del client OAuth2. Questi dettagli includono l'ID del client, il segreto del client, l'URI di reindirizzamento, i metodi di autenticazione del client, i tipi di concessione di autorizzazione e le impostazioni del client.

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
    RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID().toString())
            .clientId("client")
            .clientSecret(passwordEncoder().encode("secret"))
            .scope("read")
            .scope(OidcScopes.OPENID)
            .scope(OidcScopes.PROFILE)
            .redirectUri("http://127.0.0.1:8080/login/oauth2/code/myoauth2")
            .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
            .clientSettings(clientSettings())
            .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
}
```

Token JWT (1)

Come già anticipato, utilizzando OAuth2.0, per ogni richiesta per cui è necessaria l'autenticazione viene generato un **token**, che contiene alcune informazioni dell'utente (claims). Il client fornirà il token permettendo al server di identificare l'utente.
Vediamo un esempio di token JWT, che è stato utilizzato per l'applicazione in esame:

eyJraWQiOiJhOGRlZGNjNy0xNjFiLTQ0OTYtYTFkYS1lNGQzMjI5YzBhYjgiLCJhbGciOiJSUzI1NiJ9.
eyJzdWIiOiJzaW1vbmUuZ2FsaW1iZXJ0aUBnbWFpbC5jb20iLCJhdWQiOiJjbGllbnQiLCJuYmYiOjE3MTI1ODE3
MjcsIlNPQVNlYyI6IlNPQVNlYyIsInNjb3BlIjpbIm9wZW5pZCJdLCJpc3MiOiJodHRwOi8vbG9jYWxob3N0O
jkwMDAiLCJleHAiOjE3MTI1ODIwMjcsImlhdCI6MTcxMjU4MTcyNywiYXV0aG9yaXRpZXMiOlsicHJlc2lkZW56YSJdfQ.
YDe4YuDcnPpdRsBSJFZbCHac4TtPtqEmJ4E4nroyLj3wjhQOdblFGNtZCmaOfQbBVLYcoiUwIjCj41XhLBML
eZ4gEaaNY3tvJlkKihOhIqh3vm-zUqf9EO_h0_zmnJ7dk4z2z-wgiRafQ_z6vQkraTaWAhxo3Qt4dec4Dc8RN
mYcegCQlexe1SvIBLIkL5KbRT2Nhq5Irb50xFEoSjg4uGOCshOewZ3jkXvU0Gw5l65KoHVvbKIFbdduKfa-laKuCA
6HiuCYPVopFaJNWzF_w2IwWIIgX4-fTY7ozFvALovnjNwEIQGoZbTOvAhm0t1Cj56JdUald24s9KxPi8jHNw

Token JWT (2)

Il token si divide in 3 parti: **header** (in rosso), **payload** (in viola) e **signature** (in azzurro). Esse una volta decodificate contengono un JSON con alcune informazioni.

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "kid": "c7dbdacb-44b1-4ada-b28d-3301b5c78ad0",
  "alg": "RS256",
  "typ": "JWT"
}
```


Figura: Header decodificato

PAYLOAD: DATA


```
{
  "sub": "simone.galimberti2@studenti.unimi.it",
  "aud": "client",
  "nbf": 1712827395,
  "SOASec": "SOASec",
  "scope": [
    "openid"
  ],
  "iss": "http://localhost:9000",
  "exp": 1712827695,
  "iat": 1712827395,
  "authorities": [
    "amministratore"
  ]
}
```


Figura: Payload decodificato

Token JWT (3) - Il metodo tokenCustomizer

Il metodo tokenCustomizer definisce un OAuth2TokenCustomizer che **personalizza i token JWT** emessi. Aggiunge rivendicazioni personalizzate ai token di accesso e di identità. In particolare vengono inseriti come claim l'indirizzo email dell'utente e il suo ruolo.

```
@Bean
OAuth2TokenCustomizer<JwtEncodingContext> tokenCustomizer() {
    return context -> {
        Authentication principal = context.getPrincipal();
        Set<String> authorities = principal.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority).collect(Collectors.toSet());
        if (context.getTokenType().getValue().equals("id_token")) {
            context.getClaims().claim("SOASec", "IdSOASec")
                    .claim("authorities", authorities)
                    .claim("user", principal.getName());
        }
        if (context.getTokenType().getValue().equals("access_token")) {
            context.getClaims().claim("SOASec", "SOASec");
            context.getClaims().claim("authorities", authorities);
        }

    };
}
```

Figura: Codice Java del metodo

Token JWT (4) - La firma dei token

La terza parte del token, denominata **signature**, è il risultato di una funzione di hash 256. Essa prende in input la codifica *base64* dell'header, la concatena alla codifica *base64* del payload e il tutto viene cifrato utilizzando l'algoritmo di cifratura asimmetrico *RSA*, di cui vengono opportunamente generate le apposite chiavi pubbliche e private dai metodi mostrati in figura. Ciò aggiunge un ulteriore livello di sicurezza.

```
public static RSAKey generateRsa() {
    KeyPair keyPair = generateRsaKey();
    RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
    RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
    return new RSAKey.Builder(publicKey).privateKey(privateKey).keyID(UUID.randomUUID().toString()).build();
}


static KeyPair generateRsaKey() {
    KeyPair keyPair;
    try {
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048);
        keyPair = keyPairGenerator.generateKeyPair();
    } catch (Exception ex) {
        throw new IllegalStateException(ex);
    }
    return keyPair;
}
```

Figura: Metodi generazione chiavi RSA

RESOURCE SERVER

Introduzione

Il Resource Server è un componente chiave che protegge le risorse degli utenti e ne controlla l'accesso consentendolo solo a coloro che sono autorizzati. In parole semplici, funge da custode delle informazioni riservate e gestisce le richieste di accesso in modo sicuro e affidabile.
Nel nostro caso, il server è in ascolto sulla porta 8080.

Classe securityConfig

La pagina di login è impostata su */oauth2/authorization/myoauth2*.
Un gestore di successo personalizzato (*successHandler()*) viene utilizzato per gestire i login riusciti.
Il servizio utente OIDC (*this.oidcUserService()*) viene utilizzato per recuperare le informazioni utente dal provider OAuth2.
L'URL di logout è impostato su */logout*. Dopo un logout riuscito, gli utenti vengono reindirizzati a *http://localhost:8080/*.

```
@Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http, ClientRegistrationRepository
clientRegistrationRepository) throws Exception {

        String base_uri =
OAuth2AuthorizationRequestRedirectFilter.DEFAULT_AUTHORIZATION_REQUEST_BASE_URI;
        DefaultOAuth2AuthorizationRequestResolver resolver = new
DefaultOAuth2AuthorizationRequestResolver(clientRegistrationRepository, base_uri);
        resolver.setAuthorizationRequestCustomizer(OAuth2AuthorizationRequestCustomizers.withPkce());

        http
                .authorizeHttpRequests(requests -> requests
                        .requestMatchers("/amministratore").hasAuthority("amministratore")
                        .requestMatchers("/amministratore/api/**").hasAuthority("amministratore")
                        .requestMatchers("/docente").hasAuthority("docente")
                        .requestMatchers("/docente/api/**").hasAuthority("docente")
                        .requestMatchers("/studente").hasAuthority("studente")
                        .requestMatchers("/studente/api/**").hasAuthority("studente")
                        .requestMatchers("/personale_presidenza").hasAuthority("presidenza")
                        .requestMatchers("/personale_presidenza/api/**").hasAuthority("presidenza")
                        .requestMatchers("/").permitAll()
                        .requestMatchers("/login").permitAll()
                )
                .csrf(csrf -> csrf
                        .disable()
                )
                .oauth2Login(oauth2Login -> {
                    oauth2Login.loginPage("/oauth2/authorization/myoauth2");
                    oauth2Login.authorizationEndpoint().authorizationRequestResolver(resolver);
                    oauth2Login.successHandler(successHandler());
                    oauth2Login.userInfoEndpoint(userInfo -> userInfo
                            .oidcUserService(this.oidcUserService()));
                })
                .oauth2Client(withDefaults())
                .logout()
                .logoutUrl("/logout")
                .logoutSuccessUrl("http://localhost:8080/");
        return http.build();
    }
```

Classe successHandler

Questo metodo implementa una logica di redirect basata sulle autorità dell'utente. A seconda del ruolo dell'utente autenticato il metodo stabilisce la pagina di destinazione appropriata.
Il parametro *authentication* è un oggetto Authentication che contiene le informazioni sull'utente.
Il metodo restituisce la stringa url che contiene l'URL di destinazione determinata in base all'autorità dell'utente.

```
protected String determineTargetUrl(Authentication authentication) {
    String url = "";

    Collection<? extends GrantedAuthority> authorities = authentication.getAuthorities();

    for (GrantedAuthority grantedAuthority : authorities) {
        String authorityName = grantedAuthority.getAuthority();
        if (authorityName.equals("studente")) {
            url = "/studente";
            break;
        } else if (authorityName.equals("amministratore")) {
            url = "/amministratore";
            break;
        } else if (authorityName.equals("presidenza")) {
            url = "/personale_presidenza";
            break;
        } else if (authorityName.equals("docente")) {
            url = "/docente";
            break;
        }
    }

    return url;
}
```

API

- Amministratore
- Presidenza
- Docente
- Studente

API - Amministratore

Chiamata	Input	Output	Descrizione
/amministratore/api/utente	Nulla	Collezione di utenti	Restituisce tutti gli utenti
/amministratore/api/utente/{id}	ID	Utente	Restituisce l'utente con ID indicato
/amministratore/api/utente	Utente	Utente creato	Crea un nuovo utente
/amministratore/api/utente/{id}	ID,Utente	Utente aggiornato	Aggiorna un utente esistente
/amministratore/api/utente/{id}	ID	Nulla	Elimina l'utente con ID indicato
/amministratore/api/nomeUtente	Nulla	Nome utente	Restituisce il nome utente
/amministratore/api/utente/cambiaPassword	Nulla	Utente aggiornato	Modifica Password

API - Presidenza

Chiamata	Input	Output	Descrizione
/personale_presidenza/api/circolare	Nulla	Collezione di circolari	Restituisce tutte le circolari
/personale_presidenza/api/circolare/{id}	ID	Circolare	Restituisce la circolare con ID indicato
/personale_presidenza/api/circolare	Circolare	Circolare creata	Crea una nuova circolare
/personale_presidenza/api/circolare/{id}	ID,Circolare	Circolare aggiornata	Aggiorna una circolare esistente
/personale_presidenza/api/circolare/{id}	ID	Nulla	Elimina la circolare con ID indicato
/personale_presidenza/api/nomeUtente	Nulla	Nome utente	Restituisce il nome utente
/personale_presidenza/api/utente	Nulla	Utente	Restituisce l'utente loggato
/personale_presidenza/api/utente/{id}	ID, Utente	Utente aggiornato	Modifica Password

API - Docente/Studente

Chiamata	Input	Output	Descrizione
/docente/api/circolare	Nulla	Circolari	Recupera tutte le circolari che gli riguardano
/docente/api/circolare/{id}	ID	Circolare	Restituisce circolare con id indicato
/docente/api/nomeUtente	Nulla	Nome utente	Restituisce il nome utente
/docente/api/utente	Nulla	Utente	Restituisce l'utente loggato
/docente/api/utente/{id}	ID, Utente	Utente aggiornato	Cambia la password

Database

Circolare

#	Nome	Tipo	Codifica caratteri	Attributi	Null	Predefinito	Commenti	Extra
1	**id**	int(11)			No	*Nessuno*		AUTO_INCREMENT
2	**titolo**	text	utf8mb4_general_ci		No	*Nessuno*		
3	**descrizione**	text	utf8mb4_general_ci		No	*Nessuno*		
4	**data**	text	utf8mb4_general_ci		No	*Nessuno*		
5	**tipo**	text	utf8mb4_general_ci		No	*Nessuno*		

Utente

#	Nome	Tipo	Codifica caratteri	Attributi	Null	Predefinito	Commenti	Extra
1	**id**	int(11)			No	*Nessuno*		AUTO_INCREMENT
2	**nome**	text	utf8mb4_general_ci		No	*Nessuno*		
3	**cognome**	text	utf8mb4_general_ci		No	*Nessuno*		
4	**email**	varchar(255)	utf8mb4_general_ci		No	*Nessuno*		
5	**password**	text	utf8mb4_general_ci		No	*Nessuno*		
6	**tipo**	text	utf8mb4_general_ci		No	*Nessuno*		

PasswordEncoder (1)

Il metodo definisce un PasswordEncoder che utilizza **BCryptPasswordEncoder**.
Questo viene utilizzato per codificare le password in modo sicuro.
E' l'implementazione predefinita utilizzata da Spring Security.

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Il **client secret**, utilizzato per l'autenticazione del client verrà quindi codificato con il passwordEncoder.

PasswordEncoder (2)

- PasswordEncoder utilizza la funzione di hashing forte **BCrypt**.
- Ad esempio "antonio" viene cifrato in
- "$2a10Tjlzb8KGwwiWZ...bmrWV1UTlFykKKjWXXxS"

PasswordEncoder (3)

```
@Override
public User create(User utente) {
    String encryptedPassword = passwordEncoder.encode(utente.getPassword());
    utente.setPassword(encryptedPassword);
    return utenteRepository.save(utente);
}
```

PasswordEncoder (4)

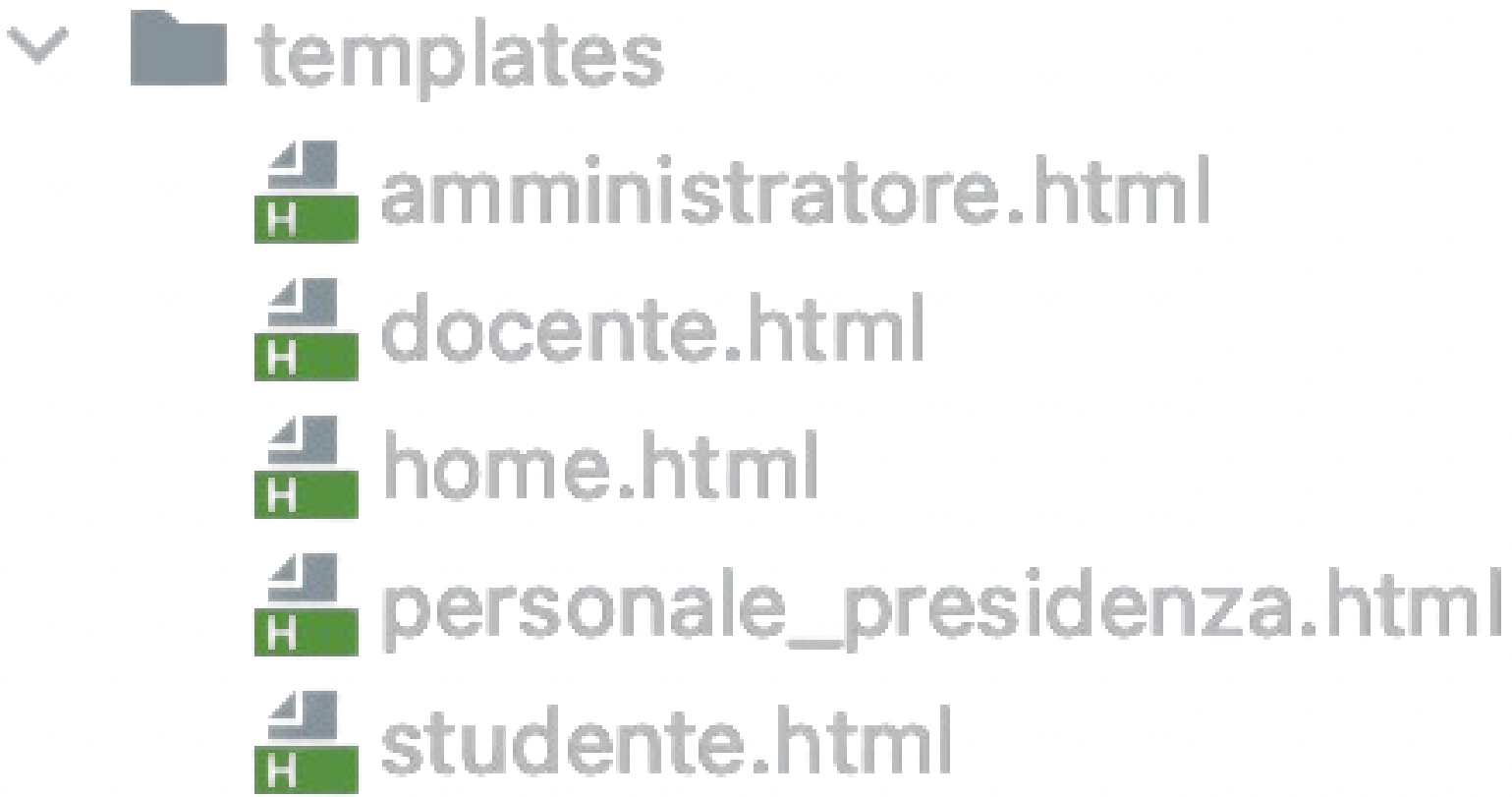
```
@Override
public Optional<User> update(int id, @RequestBody User utente) {
    Optional<User> trovaUtente = utenteRepository.findById(id);
    if(trovaUtente.isEmpty()) {
        return Optional.empty();
    }
    trovaUtente.get().setNome(utente.getNome());
    trovaUtente.get().setCognome(utente.getCognome());
    trovaUtente.get().setEmail(utente.getEmail());
    String encryptedPassword = passwordEncoder.encode(utente.getPassword());
    trovaUtente.get().setPassword(encryptedPassword);
    trovaUtente.get().setTipo(utente.getTipo());
    utenteRepository.save(trovaUtente.get());
    return trovaUtente;
}
```

CLIENT

Introduzione

Il Client rappresenta l'applicazione software che richiede l'accesso alle risorse HTTP protette.

Templates


templates
amministratore.html
docente.html
home.html
personale_presidenza.html
studente.html

Esempio - amministratore.html (1)

```
//acquisisce i dati per creare un oggetto utente che verrà inviato al Authorization server
$("#caricaUtenteBtn").on('click', function () {
    var nome;
    do {
        nome = prompt("Inserisci il nome dell'utente");
       } while(nome == '');
    var cognome;
    do {
        cognome = prompt("Inserisci il cognome dell'utente");
       } while(cognome == '');
    var email;
    do {
        email = prompt("Inserisci l'email dell'utente");
       } while(email == '');
    var password;
    do {
        password = prompt("Inserisci la password dell'utente");
       } while (password == '');
   var tipo;
    do {
       tipo = prompt('Inserisci il tipo: studente, docente, presidenza');
       } while (!tipiAccettabili.includes(tipo) && tipo !== null);
```

Esempio - amministratore.html (2)

```
var utente = {
    nome: nome,
    cognome: cognome,
    email: email,
    password: password,
    tipo: tipo
};

$.ajax({
    url: '/amministratore/api/utente',
    method: 'post',
    contentType: 'application/json',
    data: JSON.stringify(utente),
    success: function (data) {
        caricaUtente();
    },
```

Esempio frontend (1)

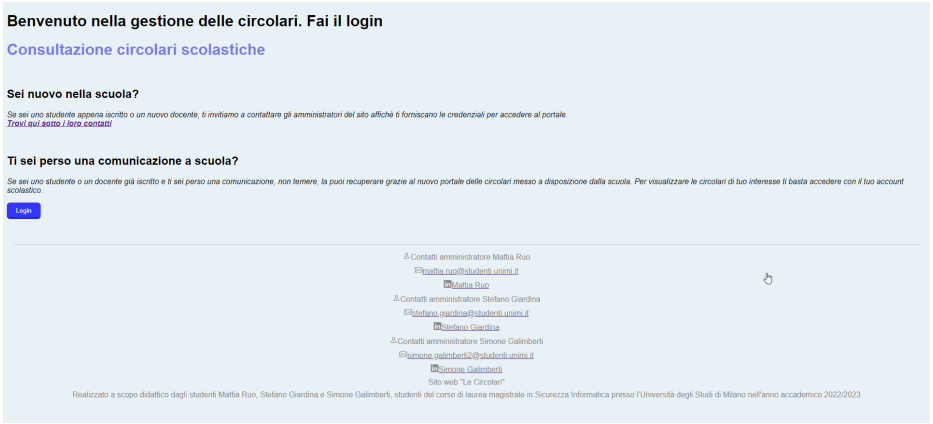


Figura: Home Page

Figura: Pagina docenti

Esempio frontend (2)

Figura: Pagina amministratori

Figura: Pagina presidenza

Conclusione

GRAZIE PER L'ATTENZIONE