

Autenticazione & Autorizzazione

Guard, Interceptor, JWT

The background features a large, semi-transparent circle with a gradient from dark purple at the top to bright cyan at the bottom. This circle overlaps a smaller, solid dark purple circle located in the lower-left quadrant. The word "GUARD" is written in a bold, white, sans-serif font, centered within the cyan-colored area of the larger circle.

GUARD

DEFINIZIONE

Le guardie sono delle interfacce fornite da Angular che, una volta implementate, ci consentono di controllare l'accessibilità di una rotta in base alle condizioni fornite nell'implementazione di classe di tale interfaccia.

TIPI DI GUARDIE

- **CanActivate**
 - controlla se l'utente può accedere alla rotta
- **CanActivateChild**
 - controlla se l'utente può accedere ai figli della rotta
- **CanLoad**
 - impedisce al modulo intero di caricarsi se non si ha accesso alla rotta.
Va implementato se la rotta è caricata attraverso il *Lazy Loading*.
- **CanDeactivate**
 - determina se l'utente può lasciare la rotta su cui si trova
- **Resolve**
 - viene utilizzato quando desideriamo assicurarci se ci sono dati disponibili o meno prima di navigare verso qualsiasi rotta

- Esempio implementazione **CanLoad**

```
@Injectable({
  providedIn: 'root'
})
export class RoleAdminGuard implements CanLoad {
  constructor(private authService: AuthService){}

  canLoad(
    route: Route,
    segments: UrlSegment[]): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

    const role = this.authService.getRole();
    if(role == "admin")
      return true;

    alert("solo gli admin possono accedere a questa rotta");
    return false;
  }
}
```

- Esempio implementazione **CanActivate**

```
@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    if(this.authService.getStatusUser())
      return true;

    alert("per vedere la home devi essere loggato");
    return false
  }
}

{path: "home",
  loadChildren: () => import('./features/home/home.module')
    .then(m => m.HomeModule), canActivate: [AuthGuard]},
```



INTERCEPTOR

DEFINIZIONE

Intercettano le response e le request delle chiamate HTTP effettuate dalla nostra applicazione al fine di manipolarle.

E' possibile registrare più di un interceptor in modo da formare una pipeline che verrà eseguita prima che parta la chiamata HTTP.

Quando effettuiamo una chiamata http potrebbe essere necessario inserire delle intestazioni HTTP da passare al server oltre ai dati.
es. validare una chiamata attraverso un token

Implementazione

Un interceptor viene creato attraverso l'interfaccia “HttpInterceptor”.

Creando un servizio che implementa quest'interfaccia ci viene richiesto di implementare il metodo “intercept”, che verrà invocato ad ogni richiesta HTTP.

Dopo aver creato questo servizio dobbiamo poi preoccuparci di registrarlo(effettuare il providing) come un qualsiasi altro servizio.



Il metodo intercept

Il metodo intercept ha due parametri in ingresso.

Il primo parametro contiene la richiesta HTTP e implementa l'interfaccia `HttpRequest`. Tale parametro ha poi una particolarità: “è immutabile”. Pertanto se dobbiamo modificare la richiesta HTTP passata dobbiamo clonarla attraverso il metodo “`clone()`” per poi modificare l'oggetto ritornato.

Il secondo parametro invece, implementa l'interfaccia `HttpHandler` e definisce il metodo `handle` a cui viene passato in input una `HttpRequest` e restituisce un `Observable`. Nel momento in cui il metodo `handle` viene invocato il controllo passerà all' interceptor successivo (se esiste), in caso contrario partirà la chiamata HTTP al server.

```
@Injectable()
export class TokenInterceptor implements HttpInterceptor {
  constructor(private authService: AuthService) {}

  intercept(req: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    const idToken = localStorage.getItem("id_token");
    alert("Ucci ucci.. sento odore di http!")

    if(idToken){
      const cloned=req.clone({
        headers: req.headers.set("Authorization",
          "Bearer "+idToken)
      });

      return next.handle(cloned);
    }else{
      return next.handle(req);
    }
  }
}
```

Providing interceptor

Per il providing vengono passati 3 parametri :

```
providers: [
  {
    provide: HTTP_INTERCEPTORS,
    useClass: TokenInterceptor,
    multi: true
  }
],
```

- 1) **HTTP_INTERCEPTORS**: identifica che si tratta di un interceptor
- 2) **useClass**: identifica il nome della classe da istanziare
- 3) **multi**: quando ha valore true indica che il token HTTP_INTERCEPTORS è un multiprovider ovvero effettua l'injection di un array di istanze

ESTRARRE DATI DAL JWT

Libreria jwt-decode

<https://www.npmjs.com/package/jwt-decode>

per salvare i dati dell'utente loggato
abbiamo la necessità di estrarli dal JWT
ricevuto dal server

grazie al metodo jwt_decode offerto dalla
lib avremmo accesso alle info contenute nel
jwt

```
getDecodedAccessToken(token: string): any {  
  try {  
    // console.log(jwt_decode(token));  
    return jwt_decode(token);  
  } catch(Error) {  
    return null;  
  }  
}  
import jwt_decode from 'jwt-decode';
```

SALVARE DATI JWT

I dati decodificati dal JWT vengono salvati all'interno del localStorage per poterne garantire l'utilizzo in tutta l'applicazione.

```
setSession(authResult: any){  
  localStorage.setItem('id_token',authResult.idToken);  
  this.setRole()  
}  
  
setRole(){  
  let user = this.getDecodedAccessToken(this.authResult.idToken);  
  let role = user.role  
  localStorage.setItem('role', role);  
}
```

LOGIN

```
realLogin(email:string, password:string) {
  const encodedCredentials = btoa(password);
  return this.http.post<{id_Token: string}>(
    'http://www.your-server.com/auth/login', {email, encodedCredentials}).pipe(tap(res => {
      localStorage.setItem('id_token', res.id_Token);}))
}
```

L'operatore Angular Tap RxJs restituisce un osservabile identico alla sorgente. Non modifica in alcun modo il flusso. L'operatore Tap è utile per registrare il valore, eseguire il debug del flusso per i valori corretti o eseguire altri effetti collaterali.

In un applicativo reale, la chiamata al server per l'autenticazione, passerà al BE le credenziali per il login con una chiamata post, aspettando di ricevere il JWT, che verrà salvato nel LocalStorage

Il metodo btoa() crea una stringa ASCII con codifica Base64 da una stringa binaria (ovvero una stringa in cui ogni carattere nella stringa viene trattato come un byte di dati binari).