# Documentazione applicazione Front-End

**Obiettivo dell'applicazione:**

Il client-side dell'applicazione è stato creato con l'obiettivo principale di offrire una Single Page Application (SPA) che consenta agli utenti, previamente registrati, di accedere all'applicazione, ridurre il carico sul Web Service abbandonando l'utilizzo di sessioni che restano aperte su quest'ultimo garantendo quindi ulteriore sicurezza.

Questo obiettivo è stato raggiunto implementando un sistema di autenticazione basato su un token JWT (JSON Web Token) ed un Refresh Token, utilizzati poi per autenticare uno user ad ogni interazione con il Web Service.

**Metodi di Autenticazione:**

L'applicazione utilizza i Token per autenticare gli utenti già registrati, al successo del Login verranno generati dal Web Service i seguenti Token:

1. **Access Token:** Questo Token viene trasmesso come stringa in un JSON, ha una scadenza limitata per migliorare la sicurezza e viene salvato nello stato dell'applicazione. Servirà ad autenticare uno User includendo il token negli Headers delle chiamate HTTPS che avverranno ad ogni interazione con il Web Service, l'ultimo avrà lo scopo di validare il Token ricevuto prima di rilasciare eventuali risposte o dati.

2. **Refresh Token:** Il Refresh Token è inviato al Client Side tramite un cookie HttpOnly, anch'esso ha una durata limitata (solitamente maggiore dell'Access Token). Questo token viene utilizzato per ottenere un nuovo Access Token quando il precedente è scaduto, per motivi di sicurezza il Refresh Token non ha la possibilità di generare altri Refresh Token.

**Chiarimenti sullo scopo dei token:**

il token JWT principale (Access Token) è quello che ci consente di interagire con il Web Service una volta che l'ultimo ne ha validato l'autenticità e la scadenza. La scadenza è settata per un periodo limitato, garantendo una finestra temporale di accesso all'applicazione con lo stesso Token. Questo impedisce ad utenti malintenzionati di avere una finestra di accesso illimitata all'applicazione nel caso in cui entrino in possesso dell'Access Token.

Il Refresh Token (anch'esso con scadenza) ha invece lo scopo di chiave per la generazione altri Access Token, questo Token è trasmesso all'interno di un **HttpOnly Cookie**, pertanto risulta **inaccessibile** tramite Javascript. Questa pratica risulta un ottimo compromesso tra sicurezza e prestazioni, offrendo una protezione da eventuali attacchi XSS (Cross-Site-Scripting) che impediscono il salvataggio del Token di refresh, evitando la possibilità di generare altri Access Token.

**Librerie Utilizzate:**

- **React:** Abbiamo sviluppato l'applicazione front-end utilizzando la libreria Javascript React per creare una SPA reattiva e performante.

- **Material UI:** Abbiamo adottato Material-UI come framework per la progettazione dell'interfaccia utente nella nostra applicazione. Material-UI è una libreria open-source basata sul design system "Material Design" di Google, ed è ampiamente utilizzata per creare interfacce utente moderne.

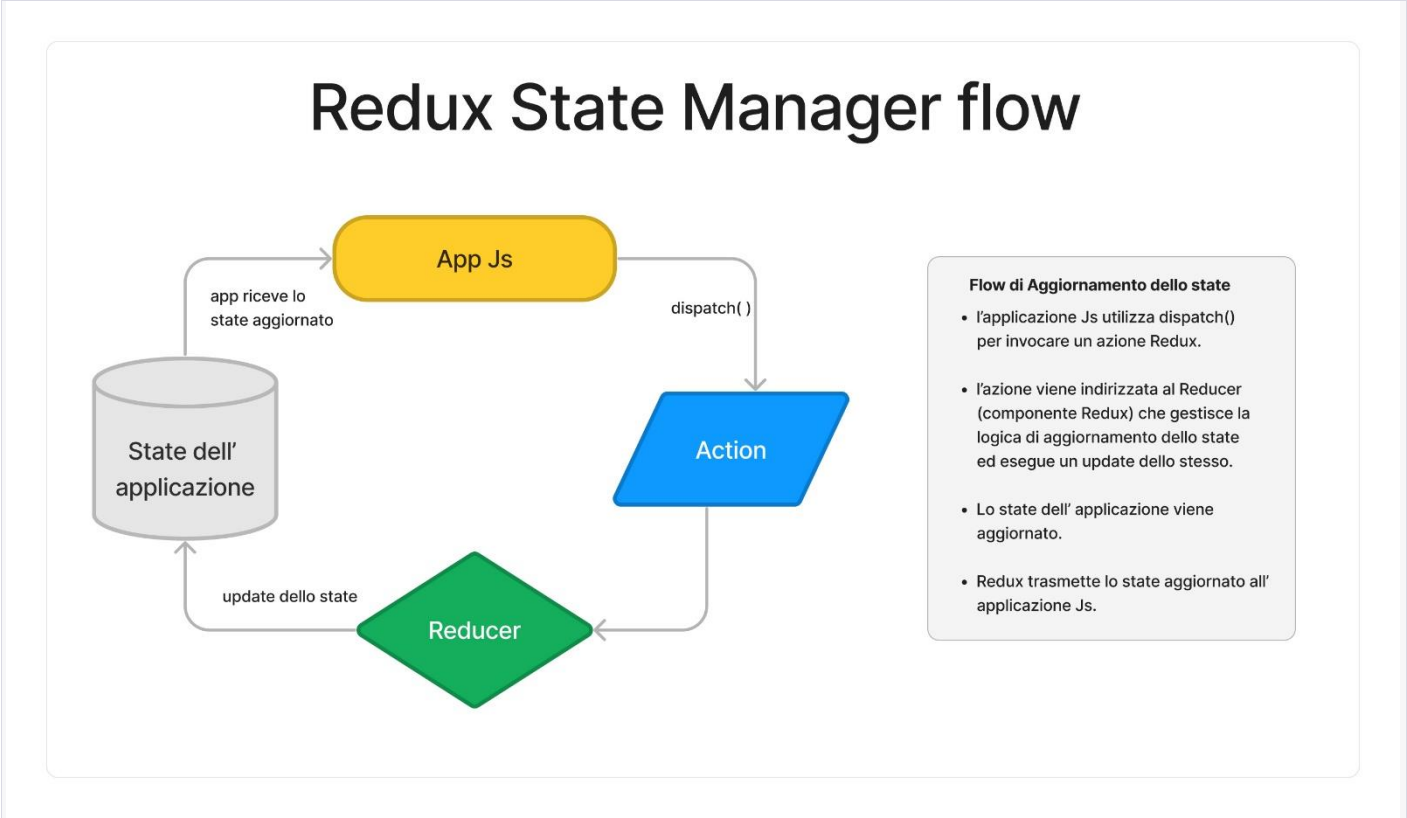
- **Redux e Redux-Toolkit:** Redux con ulteriore libreria Redux-Toolkit, sono stati utilizzati per gestire lo stato dell'applicazione in modo efficiente, consentendo un accesso sicuro e prevedibile ai dati a tutti i componenti React. Per quanto riguarda l'autenticazione, lo stato di Redux conserva i dati relativi all'utente autenticato come lo username e l'Access Token. Utilizzando lo state di Redux forniamo un ulteriore sicurezza dei dati lato client in quanto lo stesso funzionamento di Redux impedisce il rilascio e/o visualizzazione di dati conservati nel suo state se non chiamando **azioni,** invocate tramite metodo **dispatch()** della libreria stessa, solo dopo che il metodo viene chiamato con successo i dati saranno forniti all'applicazione front-end.

- **React-Router:** React-Router v.6 ci ha permesso di gestire il comportamento della visualizzazione dei componenti e delle pagine in modo reattivo senza causare il ricaricamento al cambio di un URL, inoltre ha permesso di definire rotte protette che permettono l'accesso ai soli utenti autenticati. Quando un utente tenta di accedere a queste rotte, verifichiamo lo stato di autenticazione nello state di Redux. Solo se l'utente è autenticato e quindi dispone di un Access Token, gli consentiamo l'accesso alle rotte protette. In caso contrario, l'utente viene reindirizzato alla pagina di Login per eseguire di nuovo il processo di autenticazione.

- **Redux-Toolkit Query:** Abbiamo utilizzato Redux-Toolkit-Query per semplificare la gestione delle chiamate API all'interno dell'applicazione. Con esso, abbiamo creato una API creando una query di base personalizzata delle chiamate Https mediante il metodo **fetchBaseQuery(),** aggiungendo negli Headers il cookie che contiene il Refresh Token (generato ed inviato dal Web Service all' accesso dell'utente), ed un header "Authentication" nel quale viene inserito l' Access Token, poi validato dal Web Service stesso, che deciderà il tipo di risposta. In base alla risposta la **funzione wrapper** della query di base deciderà a quale endpoint effettuare la chiamata. Nel caso in cui l'Access Token è scaduto ad esempio, il Web service risponderà con uno status 401 (non autorizzato), invece di terminare la chiamata, la Api è programmata ad eseguire una nuova chiamata all'endpoint '/refresh' che ha come scopo quello di generare un nuovo Access Token per poi salvarlo nello state e riprovare la chiamata precedente con il nuovo Token, questa volta valido.

**Vantaggi dell'Approccio Redux:**

L'uso di Redux per la gestione dello stato di autenticazione garantisce coerenza e facilità di accesso alle informazioni di autenticazione in tutta l'applicazione garantendo un ulteriore livello di sicurezza dei dati. Inoltre, riduciamo il carico del Web Service evitando di dover richiedere nuovamente l'autenticazione ad ogni navigazione tra le rotte ed evitando l'utilizzo di session sullo stesso.




## Prima apertura dell'applicazione

Al primo avvio l'applicazione si caricherà portandoci in su un 'index path' che restituisce una visualizzazione di un layout comprendente header, sezione main e footer.

Nella sezione main avviene il rendering del componente **<MainPage/>**, un componente accessibile a tutti e non protetto da routes che richiedono l'autorizzazione.

Nell' header abbiamo una sezione con *logo* a sinistra, ed una sezione di navigazione con vari link, tra cui il link "Accedi" che, se cliccato, ci indirizzerà all'URL: "/login" che farà partire il rendering del componente **<LoginPage/>**, anche questa visualizzazione non è protetta così da garantire ad ogni tipo di utente la possibilità di accedere.

## Pagina di Login

La pagina di login si presenta con un semplice form nel quale ci verrà chiesto l'inserimento dei campi "username" e "password" ed un bottone destinato al submit del form al Web Service.

**Validazione dei campi:**

È stata implementata una funzione ***handleSubmit()*** che gestisce la form validation a livello di front-end per garantire che gli utenti inseriscano dati validi. Se lo sono, richiama la funzione ***handleLogin()*** che gestisce il processo di login dell'utente

## Percorso di Autenticazione:

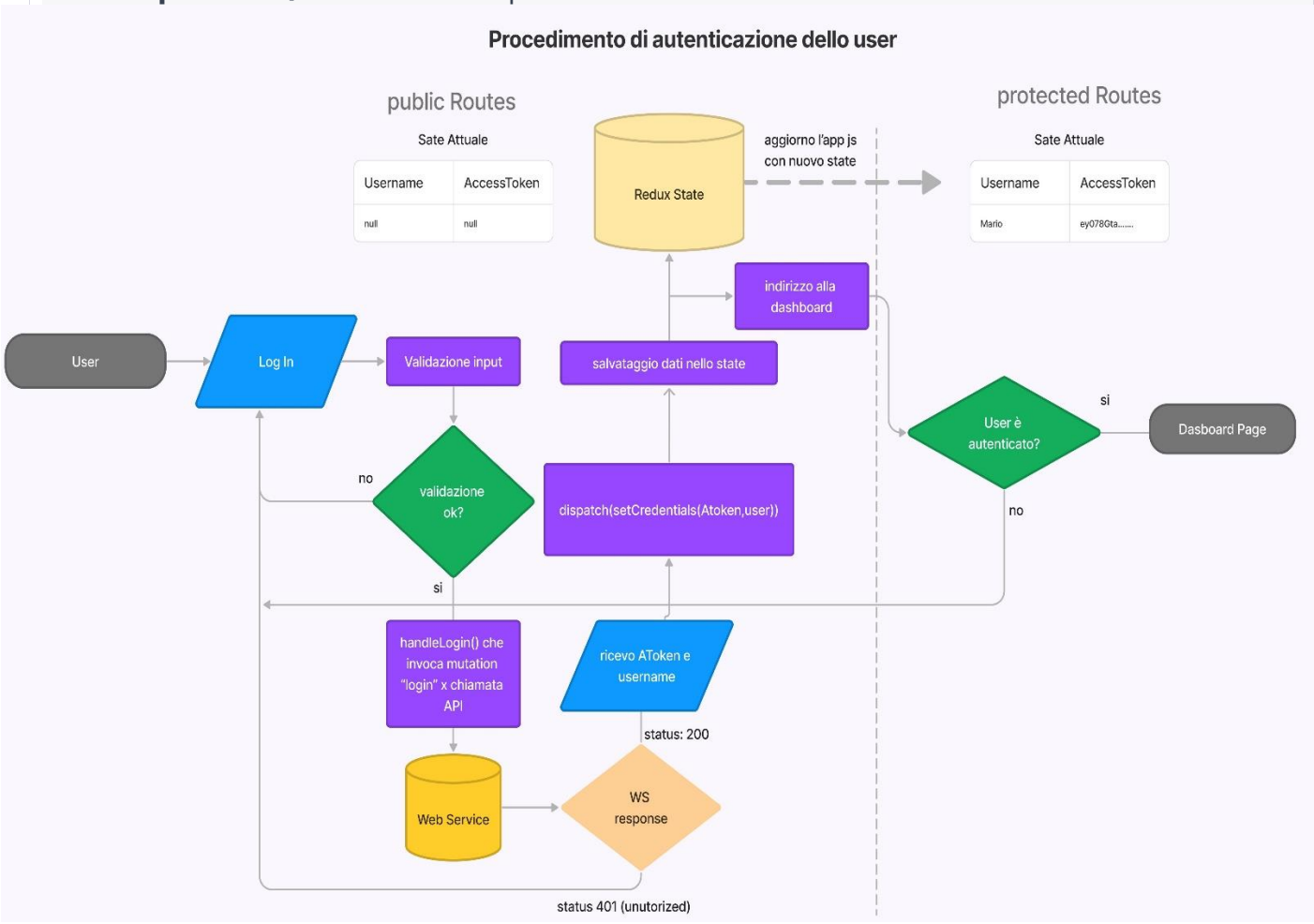
- Nella funzione ***handleLogin()***, viene generata una richiesta API di autenticazione utilizzando Redux Toolkit Query, in particolare si utilizza la mutazione del builder che chiama l'endpoint: `login`.
- Il Web Service risponde con un JSON contenente un Access Token e username dell'utente, che vengono quindi salvati nello stato dell'applicazione utilizzando ***dispatch()*** dell' azione ***setCredentials()***. Questo consente all'utente di rimanere autenticato e di accedere alle risorse protette.

## Gestione degli Errori:

- Gli errori possono verificarsi durante il processo di autenticazione. La pagina di login è progettata per gestire vari tipi di errori, inclusi quelli generati dal server API.
- Gli errori possono includere mancate risposte dal server, accesso non autorizzato o errori di autenticazione. Ogni tipo di errore viene gestito in modo appropriato, e un messaggio descrittivo viene mostrato all'utente.

## Indirizzamento alla Dashboard:

- In caso di autenticazione riuscita, l'utente viene indirizzato alle **protected routes,** tra cui come prima visualizzazione compare la pagina di dashboard, dove l'utente può accedere alle funzionalità principali dell'applicazione.
- Questo indirizzamento avviene utilizzando React Router, insieme ad un componente **<RequireAuth/>** che ha il compito di controllare se l'utente è autenticato.



## Descrizione e funzionamento della API con RTKQuery

La gestione delle chiamate al Web Service è stata implementata utilizzando una API costruita con il metodo ***createApi()*** di Redux-Toolkit-Query, abbiamo modificato la query di base di tutte le chiamate utilizzando la funzione ***fetchBaseQuery()*** sulla **baseQuery.**

```
// query di base personalizzata, le chiamate HTTPS verranno fatte con headers personalizzati che contengono l'AccessToken
const baseQuery = fetchBaseQuery({
    //url base della query, da cambiare in produzione
    baseUrl: 'https://localhost:7029',
    //< credentials:'include' >  permette di inserire automaticamente i cookies nell'header della richiesta.
    credentials: 'include',
    //creazione di headers personalizzati con metodo prepareHeaders()
    prepareHeaders: ( headers,  { getState } ) => {
        //prendo l'access Token salvato nello state
        const state: RootState = getState() as RootState;
        const token = state.auth.token
        //Salvo il token nell'header 'Authorizzation'
        if ( token ) {
            headers.set('Authorization', `Bearer ${token}`)
        }
        return headers
    }

})
```

La Query di base adesso contiene in automatico nell' Header della richiesta sia l'Access Token, che il Refresh Token contenuto nel Cookie HttpOnly generato ed inviato dal Web service al momento del login.

Per gestire la scadenza breve del Token JWT di accesso, abbiamo creato una funzione Wrapper ***baseQueryWithReauth()*** intorno alla baseQuery, in modo da rendere dinamica ed automatica la rigenerazione dell'Access Token:

```
const baseQueryWithReauth = async (args: any, api: any, extraOptions: any) => {
    //richiamo la fn baseQuery
    let result = await baseQuery(args, api, extraOptions)

    console.log('chiamata in corso',args)

    //se risponde con status 401 (non autorizzato)
    if (result?.error?.status === 401){

        //mando refresh token all' url /refresh, l'endpoint dovrebbe riconoscere il refresh token mandato con HTTPonly cookie
        const refreshResult = await baseQuery({url:'/api/Test/Refresh', method:'POST',credentials:'include'}, api, extraOptions)

        //se la risposta è avvenuta con successo quindi ha l' oggetto data
        if(refreshResult?.data){
            //prendo lo user dallo state in quanto dovrebbe essere già autenticato
            const user = api.getState().auth.user
            //salvo il nuovo accessToken nello state
            const accessToken = (refreshResult.data as RefreshData).accessToken;
            api.dispatch(setCredentials( {user, accessToken}))
            //riprovo la richiesta con il nuovo accessToken
            result = await baseQuery(args,api,extraOptions)
        //se non funziona (refreshToken scaduto)  faccio il logout
        }else{
            api.dispatch(logOut())
        }
    }
    return result
}
```

Nel caso in cui una chiamata dovesse subire risposta 401 (in questo caso significherebbe che il Web Service ha validato che il nostro Token è scaduto), l'endpoint si setta sulla rotta '/api/Test/Refresh', rotta che è destinata al controllo del Refresh Token. Se esso è ancora valido, il Web Service tornerà un JSON con oggetto data contenente il nuovo Access Token, esso verrà salvato all'interno dello state al posto di quello scaduto tramite ***dispatch()*** dell' azione ***setCredentials()*** e verrà inizializzata nuovamente la chiamata precedente con il nuovo token.

Se invece il Token di Refresh è scaduto, quindi la risposta non conterrà in questo caso dei dati, verrà eseguito il ***dispatch()*** dell'azione ***logOut()***, forzandoci ad eseguire nuovamente il processo di Login ed autenticazione dello User.

In fondo al documento troviamo l'esportazione dell' api dove possiamo notare che utilizza come baseQuery la funzione Wrapper ***baseQueryWithReauth()***.

```
export const apiSlice = createApi({
    baseQuery: baseQueryWithReauth,
    endpoints: builder => ({}),

})
```

Come endpoints si sta definendo una 'Mutazione' lasciandola di fatto libera in quanto ignetteremo le mutazioni del builder direttamente da altri file.

**Mutazioni del Builder**

Creando la Api con questa procedura la rendiamo dinamica, in quanto possiamo definire come vogliamo attraverso le mutazioni i vari endpoints ed i vari tipi di chiamata (POST/GET) come ad esempio nella mutazione di Login:

```
export const LoginSlice = apiSlice.injectEndpoints({
    endpoints: builder => ({
        login: builder.mutation({
            query: credentials => ({
                url: '/api/Test/Login',
                method: 'POST',
                body:{...credentials}
            })
        }),
    })

})


//gli hook Mutation sono generati automaticamente da RTKQuery  ( use*NAME*Mutation )
export const { useLoginMutation } = LoginSlice
```

In questo modo nella funzione ***handleLogin()*** si richiama (passando come parametro un oggetto contenente username e password) la mutazione ***Login(parametri)*** che in automatico passerà alla funzione Wrapper dell'API come parametri i valori contenuti nella query, come: endpoint, contenuto del body della chiamata e metodo.

*(metodo handleLogin() contenuto nel componente <LoginPage/>)*

```
const handleLogin = async (params: Params) => {

  try{
    //unwrap è una funzione di RTK che ci permette di utilizzare il blocco try / catch in modo che ritorni una risposta RAW
    //login() fa riferimento alla mutazione del builder => login: builder.mutation ...
    const result = await login(params).unwrap()
    //il result della fn login() sarà la risposta dal server, ovvero il JSON con accessToken, chiamiamo poi la funzione dispatch(SetCredentials())
    //salvando il contenuto di result nello state(token) + lo user
    dispatch(setCredentials({...result, user}))

    //reset degli input per non lasciare tracce
    setUser('')
    setPsw('')
    //reinderizzo alla pagina dashboard
    navigate('/dashboard')
  }catch(err:any){
    //non mi serve la gestione di 400 in quanto cè la validazione front end dove non parte il form se i campi sono vuoti
    //ne di 401 (token expired) in quanto la casistica è gestita nel wrapper della  baseQuery in api/apiSlice.ts
    console.log(err)
    if(!err?.originalStatus){
      setGenErr('nessuna risposta dal server')
      //il webService è impostato per rispondere con una risposta 500 invece di 401 (da gestire nel backend) e nel messaggio scrive "utente non trovato"
    }else if( err.originalStatus === 403 ){
      setGenErr('non autorizzato')
    }else{
      setGenErr('logIn Fallito')
    }
  }
  console.log('user ed accessToken salvati nello state')
}
```

## Test funzionalità dei JWT

Arrivati alla dashboard troviamo il primo test di richiesta da un utente già in possesso di un Access Token valido. Il pulsante ***'richiesta al server'*** invoca la funzione asincrona ***handleCall()*** che invoca la query mutation ***'request'*** abilitando l'API ad effettuare una chiamata di tipo GET all'endpoint '/api/Test' che risponde con status 200 e messaggio: ***'questa richiesta è stata autorizzata'***.

Il tutto procede finché l'access Token è valido. Dal momento in cui scade invece (ricevendo risposta 401) la API prova ad effettuare la richiesta all'endpoint ***'/api/Test/Refresh'*** che prendendo il Refresh Token dall' header "cookie" della chiamata, ne valida l'autenticità comparandolo con quello associato allo user e ne controlla la scadenza.

Se il Refresh Token supera le validazioni, l'endpoint risponderà alla chiamata con un JSON contenente il nuovo Access Token, che l'applicazione estrapolerà e salverà invocando nuovamente il ***dispatch()*** dell' azione ***setCredentials()***, per poi riprovare di nuovo la chiamata come già descritto dettagliatamente nella sezione dedicata alla API.