APPLICAZIONI INTERNET

Esercitazione n. 5 (gio 28/05)

SPA routing, login e accesso a servizi REST

-- consegna entro dom 14/6 --

Si richiede di sviluppare una applicazione front-end, basata su Angular, per la gestione di gruppi di studenti all'interno di corsi universitari che offra all'utente più viste per le diverse funzioni, scambi dati con un servizio REST remoto e permetta l'autenticazione degli utenti.

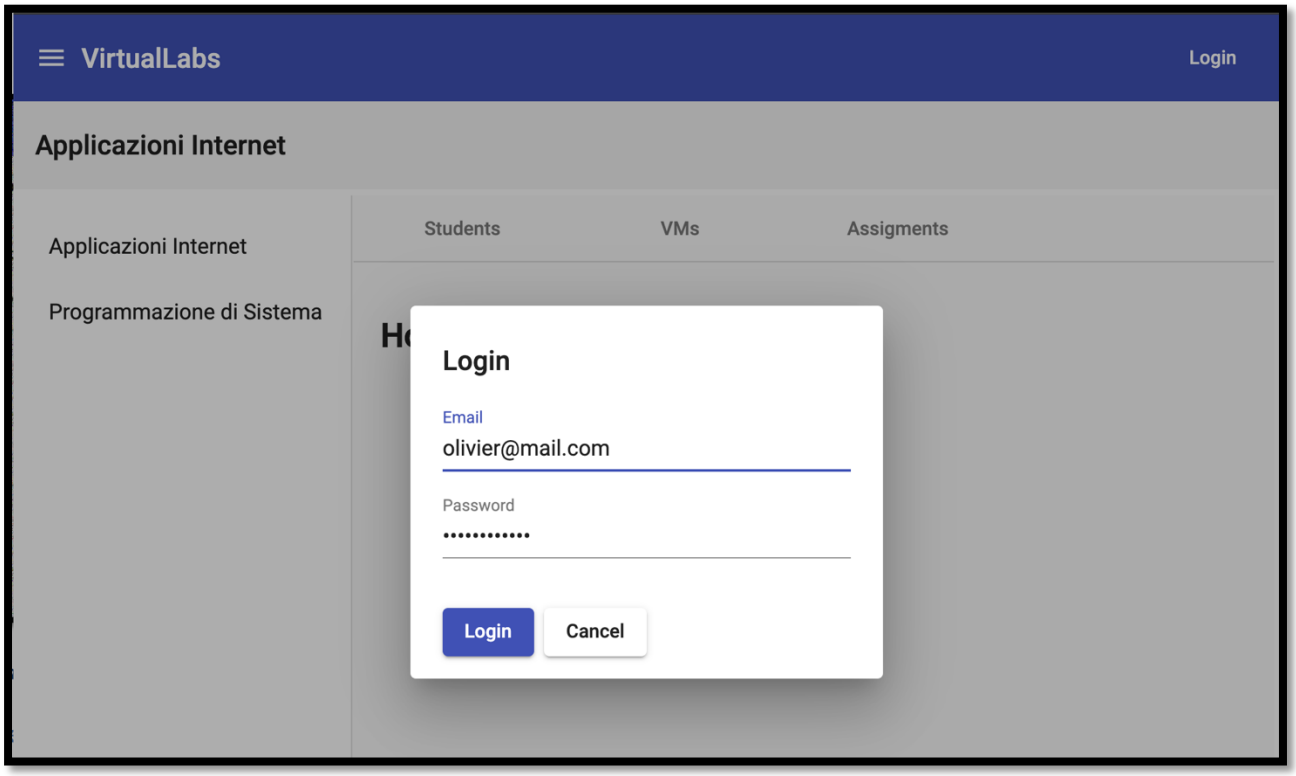

Figura 1: Vista applicazione.

Modello dei dati

In questa esercitazione utilizzeremo come server REST il programma json-server che ci offre con poco sforzo una interfaccia (abbastanza) standard per un servizio REST collegato ad un DB serializzato in formato JSON.

Va fatto notare che json-server non gestisce le relazioni molti-a-molti e anche nelle relazioni uno-a-molti non ammette la cardinalità zero. Dovremo quindi introdurre qualche semplificazione nel modello dei dati. Ad esempio: ogni studente può essere iscritto ad uno e un solo corso.

Segue un esempio dei dati e delle collezioni che andremo a rappresentare.

```
{
  "students": [ { "id": 1, "serial": "265373", "name": "ALICINO", "firstName":
"GIUSEPPE", "courseId": 1, "groupId": 0 } ],
  "courses": [ { "id": 1, "name": "Applicazioni Internet", "path": "applicazioni-
internet" } ],
  "groups": [ { "id": 1, "name": "Calvary" }  ]
}
```

Si fa notare che:

- ogni collezione è identificata da un nome plurale, e.g. <course>s, ed è un array di oggetti
- ogni oggetto ha un identificativo univoco con etichetta "id" (che parte da 1)
- la relazione con un oggetto di un'altra collezione è espressa da un campo denominato con il nome della collezione al singolare e il postfisso "Id", e.g. <course>Id, a cui viene associato il numero del campo id dell'oggetto a cui fa riferimento la relazione (ad esempio lo studente con id 1 che è iscritto al corso 1 ha campo courseId uguale a 1.
- dove la relazione è prevista, ma non è attiva (lo studente non è iscritto a nessun corso e non fa parte di nessun gruppo, si usi il valore 0.

Si faccia riferimento a [2] per maggiori informazioni anche sulle interrogazioni degli endpoint REST. Ad esempio:

- x le relazioni, se si volessero recuperare tutti gli studenti iscritti al corso con id 1 si interrogherebbe l'endpoint /course/1/students
- se, insieme ai dati propri dello studente si volesse, far "risolvere" (embeddare) anche le informazioni sul gruppo si può utilizzare la sintassi: /students?_expand=group (notare che group è singolare e students plurale) e si otterrebbe anche, direttamente, il nome del gruppo (altrimenti si avrebbe solo l'id, da groupId)

Inizialmente, come per l'esercitazione precedente, le collezioni non saranno altro che array "embeddati" nel codice dell'applicazione stessa, prima in AppComponent e poi, via via, in altri componenti, nel service e, solo verso la fine le "muoveremo" sul server REST. Si aggiunga qualche altro studente per popolare l'autocomplete e la tabella così da poterli testare.

Realizzazione dell'applicazione

1. Per procedere all'introduzione della funzionalità di routing è innanzitutto necessario "scorporare" quanto realizzato nel laboratorio precedente dal componente AppComponent (che è quello caricato in fase di boostrap e che rappresenta in un certo qual modo la *radice* della nostra applicazione). Una volta "scorporato" il nuovo componente potrà essere instanziato soltanto nella vista in cui serve (quella per la label Students), mentre in corrispondenza delle altre label verranno instanziati altri componenti.

- Si crei un nuovo componente custom denominato StudentsComponent (e i tre file associati students.component.{html,css,ts}) con all'interno il codice necessario alle funzionalità implementate "a partire" (cioè internamente) alla prima tab, che è stata denominata label="Students" (cioè il campo input con autocomplete e bottone add, più la tabella e il bottone delete).

- Si crei questo nuovo componente in una cartella teacher (perché fa parte delle viste docente). Si può utilizzare il comando `ng generate component teacher/students --flat --module` che crea i file necessari e importa il componente in app.module.ts
- Si rimuova il codice HTML dalla tab "students" e lo si collochi nel file students.component.html. Si sostituisca il codice rimosso con il tag HTML associato al nuovo componente creato (proprietà selector dell'annotazione @Component, che dovrebbe essere 'app-students', e.g. <app-students></app-students>).
- Si sposti anche il codice css e js (metodi e proprietà del componente) necessario per far funzionare il nuovo componente.

2. Per meglio strutturare l'applicazione è ancora possibile scomporre il componente appena creato in due parti: a) un componente che possiamo definire "container", che cioè rappresenta quella parte della vista "instanziata" dal router che si occupa di recuperare lo stato associato alla vista stessa (il DB degli studenti e gli studenti già iscritti al corso selezionato); b) un componente che definiamo "presentational" che non "partecipa" allo stato dell'applicazione, ma ha il solo compito di realizzare l'UI e la logica per la gestione dell'autocomplete e della tabella (questo componente interagirà con lo stato dell'applicazione solo tramite la sua interfaccia @Input/@Output). Il componente container conterrà il componente presentational.

- Si crei un ulteriore componente "container" denominato StudentsContComponent che conterrà le informazioni sul DB studenti e sugli studenti iscritti, informazioni che passerà al componente StudentsComponent attraverso il property binding.
  - Lo si crei con il comando ng generate component teacher/students-cont --flat –module
  - Si rimuovano dal componente StudentsComponent i due array che contengono l'elenco degli studenti iscritti e l'elenco di tutti gli studenti.
  - Si sostituisca in app.component.html il tag del nuovo componente 'app-students-cont' (al posto di app-students) e si proceda a modificare students-cont.component.html in modo da instanziare 'app-students' e passare i dati necessari attraverso il property binding (e.g. `[enrolledStudents]="enrolledStudents"`)
  - Se ne verifichi il corretto funzionamento

3. Come spiegato a lezione il routing viene utilizzato per navigare all'interno delle viste della nostra applicazione, ma allo stesso tempo ne rappresenta parte dello stato (es. un link /teacher/corso/applicazioni-internet/students ci dice che siamo nella vista students del docente, ma anche che il corso visualizzato è applicazioni-internet). Si crei quindi il modulo per la gestione delle route e l'instanziazione del componente corrispondente all'interno (o meglio subito dopo) il tag <router-outlet> che si colloca nel layout html del primo componente AppComponent.

ATTENZIONE. In questa fase introdurremo una semplificazione nella gestione del routing. Supporremo cioè di avere un solo corso da gestire: *Applicazioni Internet*. Corso che avrà come titolo "Applicazioni Internet", come path associato "applicazioni-internet" e come id il valore 1. Si creino pure nella sidenav altre voci con nomi di corsi differenti, ma in questo frangente non li gestiremo ancora.

- Si crei il modulo per la gestione del routing e le prime routes facendo riferimento all'esempio L11-angular-routing-E30-mock-service
  - Si crei il file src/app/app-routing-module.ts che conterrà le routes e lo si importi nel file app.module.ts (di seguito il dettaglio dell'annotazione @NgModule)

```
@NgModule({
  imports: [RouterModule.forRoot(routes, { enableTracing: false } )],
  exports: [RouterModule]
})
```

- Si crei un componente ausiliario che funga da pagina iniziale, HomeComponent, uno da visualizzare quando si provi ad accedere ad un indirizzo non valido/definito, PageNotFoundComponent (si realizzi pure ciascun componente tutto in un solo file .ts con css e html inline) e un altro per l'altra tab (tab che chiameremo VMs e il cui componente si chiamerà VmsContComponent, sempre nella cartella teacher).
- Si includa il <router-outlet> nell'html di app.component.html e si definiscano alcune route:
    - url *home* mappato sul componente *HomeComponent*, (default)
    - teacher/course/applicationi-internet/students: StudentsContcomponent
    - teacher/course/applicationi-internet/vms: VmsContcomponent
    - per tutto il resto non matchato: PageNotFoundComponent
- Si predispongano alcuni link per la navigazione come:
    - /home cliccando sul titolo "VirtualLabs"
    - teacher/course/applicazioni-internet/students cliccando sul corso 'Applicazioni Internet' nella sidenav
    - teacher/course/applicazioni-internet/students cliccando sul tab "Students"
    - teacher/course/applicazioni-internet/vms cliccando sul tab "VMs"
- ATTENZIONE: per far funzionare correttamente le tab ed il routing è necessario modificare l'HTML e sostituire il tab-group con <nav mat-tab-nav-bar> e <a mat-tab-link> (si veda anche [1]). Ricordarsi di utilizzare l'attributo routerLink per l'assegnazione dei link da cliccare per la navigazione.

4. Prima di poter realizzare il service incaricato della gestione degli studenti occorre migliorare ancora un po' l'architettura dell'applicazione. Infatti non vogliamo che il componente "presentational" debba interagire con il service, ma piuttosto delegare questo compito al componente "container". Occorre quindi che lo StudentComponent segnali allo StudentContComponent ogni modifica da effettuare: cioè aggiungere o rimuovere studenti dal corso. La modifica verrà richiesta dallo StudentContComponent al service e lo stesso StudentContComponent passerà i dati aggiornati allo StudentComponent per la visualizzazione e le successive modifiche. In questo modo il componente "presentational" è maggiormente indipendente e svincolato dal contesto e quindi riutilizzabile.

- Si modifichi il component StudentComponent per delegare al componente esterno le modifiche sulla base dati (che per il momento è ancora un array embedded nel codice e gestito dallo StudentContComponent):
    - All'atto di aggiungere uno studente tra gli iscritti al corso occorre invocare il metodo .emit su un EventEmitter passando come dato lo studente che si vuole inserire (per analogia con il caso della cancellazione, sedi oltre, si può scegliere di passare un array di studenti con un solo studente)
    - All'atto di eliminare degli studenti dagli iscritti al corso si invoca di nuovo il metodo .emit di (un altro) EventEmitter passando l'array con gli studenti da rimuovere
    - Il componente esterno intercetterà gli eventi tramite l'event binding e provvederà a modificare la lista degli studenti iscritti

- ATTENZIONE. Affinchè lo StudentComponent si accorga della modifica all'array degli studenti iscritti è necessario che:
    - L'array venga ricreato (e non solo modificato) e riassegnato (si modifichi cioè il puntatore all'array)
    - Lo StudentComponent implementi un *setter* in corrispondenza del "campo" che mappa l'array degli studenti iscritti all'interno del quale modifica l'array della MatTableDataSource. Si veda anche la documentazione sul sito di Angular [3].

```
@Input() set enrolledStudents( students: Student[] ) {
 // your code here
})
```

5. Implementiamo ora un *mockup* service a cui delegare la gestione dei dati (modello) dell'applicazione. Ci si limiti, in questo passo, a spostare la gestione dell'array embedded dal componente StudentContComponent al service, solo al punto successivo si dialogherà con un server vero.

- Si crei una classe *service* denominata StudentService per gestire le interrogazioni in merito agli studenti e la loro iscrizione ai corsi (al momento sempre uno solo).
- Nella cartella src/app/service generare il service in questione con il comando `ng generate service services/student --flat`
- Si muova all'interno del service la struttura dati _locale_ che rappresenta la collezione degli studenti.
- Si implementino i metodi per interagire con il componente e la collezione ricordandosi di ritornare, in questo caso forzatamente (così ci si uniforma a quando si avranno richieste http vere) degli observable (si utilizzi per questo scopo l'operatore of) [4]:
    - create: per creare un nuovo elemento
    - update: per aggiornarne uno esistente
    - find: per recuperarne uno dato l'id univoco
    - query: per ritornare la collezione
    - delete: per eliminare un elemento dato l'id univoco
- Si aggiungano anche il metodo (o i metodi) per:
    - Iscrivere e dis-iscrivere uno o più studenti ad un corso (e.g. updateEnrolled). Si ricorda che l'iscrizione al corso corrisponde a settare il campo courseId all'id del corso e la dis-iscrizione corrisponde a settare lo stesso campo a 0.
    - Recuperare la lista degli studenti iscritti ad un corso
- Se ne verifichi il corretto funzionamento nella vista "Students"

6. Si provveda ora a modificare il service per dialogare con il servizio REST vero invece che con i dati embedded al suo interno. Si vedano a questo scopo le slides Angular-HttpService nella sezione "Accessing a REST Endpoint" e gli esempi corrispondenti, tra cui L07-angular-httpservice-E60-rest-crud-example

- Ci si ricordi di aggiungere il modulo HttpClientModule nel file app.module.ts (sia come classe sia come modulo) e di importare HttpClient da '@angular/common/http' nelsService.
- N.B. Si presti attenzione a tipizzare i dati ricevuti dal server nelle chiamate HTTP ed eventualmente trasformarli in istanze della classe Student dove necessario.
- Dopo ciascuna modifica si ricarichi la lista degli studenti iscritti dal server per averla aggiornata (in alternativa si può ritenere di "risparmiare" al server la richiesta della

collezione dopo le modifiche, ma in questo caso non è necessario farlo perché stiamo realizzando un client "stateless", che cioè non si mantiene una copia dei dati in locale, o meglio, non una copia che aggiorna/modifica in locale).

- N.B. Nel caso di modifiche (es. dis-iscrizione dal corso) di più studenti (ne vengono selezionati N nella tabella e si chiede di rimuoverli), si presti attenzione a come le richieste vengono effettuate, se in parallelo o in serie. Si provi ad effettuarle in serie e a ricaricare la collezione solo dopo che sono state effettuate tutte le modifiche (idem nel caso parallelo, si aspetti che sia ritornata l'ultima modifica). Si veda a tal proposito (oltre alle slides) anche questo riferimento [5].

7. Si implementi ora la funzionalità di login per permettere solo all'utente loggato di accedere alle viste del docente. Per la gestione lato-server dell'autenticazione utilizziamo un fork di json-server denominato json-server-auth [6] che permette di gestire l'autenticazione tramite JWT e una forma molto semplice di autorizzazione. Si utilizzi quindi questo tool al posto di json-server.

- Si aggiunga al database JSON una collezione utenti con un utente demo come segue che avrà email/username olivier@mail.com e password bestPassw0rd. Ulteriori utenti potranno essere aggiunti tramite l'endpoint /register come indicato nella pagina del tool.

```
"users": [
  {
    "email": "olivier@mail.com",
    "password":
            "$2a$10$9SFIVF0tDrwRoUlZOo9Q0urEs0cnpBXu9RWGf/TxuHjFzDteQhvDW",
    "id": 1
  }
]
```

- L'autenticazione è gestita dall'endpoint /login come indicato nella documentazione del tool. Se ha successo viene ritornato un token (accessToken) che ha validità di un'ora e dovrà essere salvato in memoria e incluso nell'header delle richieste per avere accesso alle API protette (e.g. /students, /courses, ecc.). N.B. Il tool json-server-auth non gestisce al momento refresh token.
- L'autorizzazione è invece gestita specificando, in un file a parte, come devono essere protette le routes delle API REST. Noi ci limiteremo a definire quali route sono accessibili da chiunque, e quindi non devono essere specificate, e quali solo dagli utenti autenticati (che hanno fatto il login). La definizione delle routes, in un file virtuallabs_routes.json può essere come segue (dove le route *students*, *courses*, *groups* sono accessibili solo agli utenti autenticati, 660, se ne controlli il significato sul sito del tool [6]):

```
{ "users": 600, "students": 660, "courses": 660, "groups": 660 }
```

- Il server può essere lanciato con il comando: `npx json-server-auth virtuallabs.json -r virtuallabs_routes.json`.

8. Provvediamo quindi a creare l'interfaccia per il login e il logout degli utenti a partire dalla landing page dell'applicazione. Si utilizzerà un bottone posto nella toolbar che contiene il titolo, allineato al margine di

destra, che indicherà login/logout a seconda della necessità e, in caso di login, farà aprire una finestra popup/modale di dialogo per l'inserimento di username e password. Quando l'utente non è loggato si visualizzi il bottone di Login, dopo che ha effettuato il login e per tutta la durata di validità del token, si visualizzi il bottone Logout.

- Facendo riferimento alle indicazioni presenti sul sito di angular material per la toolbar si definisca un <button> allineato a destra che visualizzi l'etichetta "Login" (si utilizzi, come indicato uno span "example-fill-remaining-space" per gestire l'allineamento [7]
- Per realizzare la *dialog window* si seguano le istruzioni di Angular Material Dialog [8]. Si tratta di:
    - Importare il MatDialogModule in app.module.ts
    - Creare un file auth/login-dialog.component.ts in cui scrivere il codice di un LoginDialogComponent associato ad un selettore <app-login-dialog>
    - ATTENZIONE: Si noti che i componenti di tipo dialog devono essere anche dichiarati in app.module.ts in un vettore *entryComponents* (oltre che nel vettore *declarations*)
    - Si crei anche un auth/login-dialog.component.html con il codice HTML del componente per il login. A questo proposito si faccia riferimento ad una struttura base come in [9] dove però è necessario prevedere nel mat-dialog-content dei mat-form-field per inserire *email* e *password*, oltre a dei tag mat-error per segnalare quando il form non è valido (e.g. email vuota o non conforme, password vuota o di lunghezza inferiore ad un numero minimo di caratteri)
    - N.B. Per la creazione del form ci si assicuri di importare il FormsModule e il ReactiveFormsModule in app.module.ts e, per il campo password, ci si assicuri di indicare type="password" per non visualizzare i caratteri digitati.
    - Il form deve prevedere un pulsante di Login e uno di Cancel. In caso di Login occorre verificare che il form sia valido e se è così invocare il metodo di login sul *service* di autenticazione, che andremo a realizzare, passando username e password.
- Si realizzi il *service* di autenticazione nel file auth/auth.service.ts chiamandolo AuthService
    - Il servizio di autenticazione deve prevedere almeno due metodi: login e logout
    - Il metodo login deve interrogare l'endpoint /api/login con una POST per verificare username e password (si veda [6])
        - In caso di successo, quindi quando si risolve l'observable della richiesta http, deve essere memorizzato il JWT localmente in memoria e modificato il bottone in alto a destra da Login a Logout
            - E' possibile usare l'api *atob* di JS per decodificare il Base64 del JWT (e.g. `JSON.parse(atob(authResult.accessToken.split('.')[1]))`)
        - In caso di insuccesso si segnala l'errore nella dialog
    - Il metodo di logout può solo cancellare il token lato client e cambiare il bottone da Logout a Login
    - ATTENZIONE. Si noti, come da [8], che un componente può utilizzare un oggetto MatDialogRef sia per aprire la finestra di dialogo sia per essere notificato quando viene chiusa e ricevere un valore di ritorno in base al quale effettuare determinate azioni o no.

- Potrebbe essere utile definire dei metodi per controllare se un utente è loggato o no verificando la presenza del token e la sua validità temporale (la libreria moment.js, momentjs.com, può essere utile per lavorare con le date). Così come la creazione di un *model* per l'oggetto User.

9. Si invitano gli studenti a provare ad usare la funzionalità di proxy dell'ambiente di sviluppo di angular come indicato nelle slides "question-and-answers-20200518.pdf" e nella corrispondente lezione. Volendo si possono anche generare una sslKey ed un sslCert per lavorare su HTTPs (come si dovrebbe fare con i token JWT). La soluzione "proxy" fa sì che il server REST sia presentato come all'interno della "same origin" dell'applicazione web e quindi non abbia necessità di aprire le sue API con i CORS.

10. Tramite l'applicazione angular non è però ancora possibile accedere alle API protette perché le richieste AJAX non presentano l'header Authorization con l'accessToken corretto.

- Si verifichi quindi che, con un tool come Postman, postman.com, o altro strumento per gestire le richieste HTTP e impostare correttamente l'header Authorization, non è possibile accedere alle API protette senza prima aver acquisito il token JWT, e viceversa, una volta acquisito si venga riconosciuti e venga garantito l'accesso.
- Si implementi un *interceptor* in angular per far sì che ogni richiesta HTTP verso le API del server REST venga arricchita con l'header 'Authorization' contenente 'Bearer ' + l'accessToken (si vedano a questo scopo le slides Angular-HttpService). ATTENZIONE: si rammenti la particolarità nella registrazione degli interceptors nell'AppModule attraverso HTTP_INTERCEPTORS.

11. Un ulteriore passaggio, che non ha a che fare con la sicurezza, ma con l'usabilità lato client, è quello di impedire all'utente non loggato di accedere a delle viste che non verrebbero popolate perché non ha in quel momento le credenziali per accedere alle API REST lato server. In questo caso le route "sotto" /course. N.B. L'ideale sarebbe prevedere ancora un livello sopra course, distinto per docente e studenti e trasformare le rispettive route in /teacher/course/:courseId/students e /student/course/:courseId/vms così da poter distinguere anche i due ruoli con dei controlli separati per le viste "sotto" /teacher e quelle "sotto" /student. Ma si lascia questo compito per il progetto.

- Si crei una authorization guard denominata *AuthGuard* attraverso il comando `ng generate guard -d auth/auth --flat --implements CanActivate`
- Si ri-strutturino le routes in modo da avere le route da proteggere come figlie di una route "courses" da proteggere con l'AuthGuard appena creata. ATTENZIONE: non si associ un componente a questa route padre, altrimenti "nasconde" quelli instanziati dai figli che diventano "nested".
    - Se ne verifichi il funzionamento (avendo AuthGuard che restituisce sempre true) nel navigare l'applicazione (tab students e vms)
- Si completi il codice dell'AuthGuard con una funzione checkLogin(url: string) che:
    - Verifichi tramite l'AuthService se l'utente è loggato
    - Se non è loggato salvi l'indirizzo a cui voleva fare accesso (url) in modo da poterlo riportare lì dopo il login, e lo indirizzi alla pagina di login

- Avendo fatto la pagina di login come una dialog che viene aperta da AppComponent (che è associato alla route /home), ma solo a seguito del click sul bottone Login … occorre inventarsi una soluzione un po' più elaborata per "agganciare" il tutto al routing. Faremo in modo che l'accesso all'indirizzo /home/ apra la dialog di login se nell'url è contenuto il parametro doLogin=true
    - Si modifichi il bottone login in modo che non chiami direttamente un metodo dell'AppComponent, ma rediriga all'url /home?doLogin=true (si usi routerLink e queryParams)
    - Nell'AppComponent ci si registri alle modifiche della route (ActivatedRoute), in particolare ai parametri queryParams per essere richiamati al loro cambiamento e aprire la dialog per il login quando il parametro doLogin è uguale true. (se si utilizza la subscription ci si ricordi la unsubscribe)
    - ATTENZIONE: si verifichi che se nella dialog del login si sceglie "cancel" poi si rischia che la finestra non venga più riaperta se si prova a fare login o accedere ad una route protetta. Perché? … Consiglio: se l'utente sceglie "cancel" non si rimanga nella route /home?doLogin=true, ma si vada alla route /home (anche semanticamente più corretto)

Indicazioni per la consegna

L'elaborato deve essere consegnato come archivio ZIP con nome come segue <cognome-nome>-lab05-v<N>.zip, dove N è la versione dell'elaborato consegnato. Nota: non so se si possano eliminare consegne precedenti, nel caso non fosse possibile la prima avrà N=1 e le altre avranno valori di N interi successivi. Nota: non si faccia l'upload della cartella *node_modules* con le librerie!!!

Predisporre un file testuale denominato come README.TXT con:

- Nome, cognome e matricola
- istruzioni per l'avvio (dovrebbe essere sufficiente fare npm install, npm start ed aprire il browser su localhost:4200)
- indicazioni per la verifica delle funzionalità dell'applicazione sviluppata
    - dove cliccare, cosa scrivere, cosa aspettarsi, eventuali password

Si fornisca anche uno (o più) screenshot dell'interfaccia realizzata.

Riferimenti

[1] Angular Material: Tabs and navigation – https://material.angular.io/components/tabs/overview#tabs-and-navigation

[2] JSON-server - https://github.com/typicode/json-server e https://jsonplaceholder.typicode.com/

[3] Angular Component Interaction - https://angular.io/guide/component-interaction#intercept-input-property-changes-with-a-setter

[4] RxJs of - https://www.learnrxjs.io/learn-rxjs/operators/creation/of

[5] Angular University, Comprehensive Guide to Higher-Order RxJs Mapping Operators - https://blog.angular-university.io/rxjs-higher-order-mapping/

[6] JSON-server-auth - https://github.com/jeremyben/json-server-auth

[7] Angular Material – Toolbar – Positioning toolbar content - https://material.angular.io/components/toolbar/overview#positioning-toolbar-content

[8] Angular Material – Dialog - https://material.angular.io/components/dialog/overview

[9] Angular Material – Dialog – Dialog content: https://material.angular.io/components/dialog/overview#dialog-content