valtech_

Labs Angular

Sommaire

Nous verrons au cours des travaux pratiques les points suivants :

Objectifs

Le labs a pour objectif de vous présenter les différentes fonctionnalités d'Angular. Au programme :

- Mise en place de l'environnement avec `@angular/cli`,
- Création d'un composant chronomètre et d'un Pipe,
- Communication entres les composants d'Angular,
- Développer un service mocké,
- Création d'un formulaire de saisie,
- Mise en place du routage.
- Utilisation des services Angular pour communiquer avec le backend.

TP 1

> Installation de l'environnement

Prérequis

Vérifiez que vous avez les éléments suivants d'installés sur votre poste :

- Node v6 ou plus avec la commande npm –v,
- Git, nous l'utiliserons pour récupérer le projet initial,
- Webstorm (ou un autre IDE)

Installation

Initialisation du projet

Nous allons utiliser l'outil `@angular/cli` pour générer le projet. Placez-vous dans votre workspace et lancez les commandes suivantes :

```
npm install -g @angular/cli
ng new tp-angular
cd tp-angular
```

> Une fois le projet créé vous pouvez l'ouvrir dans votre IDE.

Pour lancer le serveur de développement :

```
ng serve
```

Installation d'Angular Material

Nous allons utiliser Angular material pour créer les interfaces de nos applications.

Lancer la commande suivante:

```
npm install --save @angular/material
npm install --save @angular/animations
```

Puis rajouter dans le fichier `app.module.ts` la dépendence du module comme suivant :

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { MaterialModule } from '@angular/material';
// other imports
@NgModule({
  ...
  imports: [
      BrowserAnimationsModule,
      MaterialModule
  ],
  ...
})
export class AppModule { }
```

En complément vous pouvez rajouter la feuille de style suivante dans la page index si vous voulez utiliser les icônes de la librairie Material :

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
```

Enfin dans le `styles.css`, ajouter le thème :

```
@import '~@angular/material/prebuilt-themes/indigo-pink.css';

body {
    padding: 0;
    margin:0;
    font-family: Roboto,"Helvetica Neue",sans-serif;
}
```

Générer des Composants, Directives, Pipe, etc...

Vous pouvez utiliser la commande `ng generate` (ou `ng g`) pour générer des composants Angular :

```
ng generate component my-new-component
ng g component my-new-component # using the alias

# components support relative path generation
# if in the directory src/app/feature/ and you run
ng g component new-cmp
# your component will be generated in src/app/feature/new-cmp
# but if you were to run
ng g component ../newer-cmp
# your component will be generated in src/app/newer-cmp
```

Voici la liste des commandes possibles pour générer une fonctionnalité d'Angular:

Génération	Usage
Component	`ng g component my-new-component`
Directive	`ng g directive my-new-directive`
Pipe	`ng g pipe my-new-pipe`
Service	`ng g service my-new-service`
Class	`ng g class my-new-class`
Guard	`ng g guard my-new-guard`
Interface	`ng g interface my-new-interface`
Enum	`ng g enum my-new-enum`
Module	`ng g module my-module`

> Correction du TP : tp1-solution

TP 2

> Le composant Chronomètre

Nous allons dans ce TP créer un composant Chronomètre. Nous verrons ainsi comment créer un composant ainsi que la création d'un Pipe pour formater l'affiche du composant.

Notre premier composant

Dans le dossier `src/app`, créer un dossier `chrono`. Ce dossier contiendra les fichiers `html`, `css` et `ts`.

Voici la sctructure cible du projet :

```
src/app
├── app.component.css
├── app.component.html
├── app.component.spec.ts
├── app.component.ts
├── app.module.ts
└── chrono
    ├── chrono.component.css
    ├── chrono.component.html
    └── chrono.component.ts
```

Vous devez donc créer les fichiers nécessaires au componsant Chrono !

Ensuite éditez le fichier `chrono.component.ts` et collez le contenu suivant :

```
import { Component, OnInit } from '@angular/core';

@Component({
    selector: 'chrono',
    templateUrl: './chrono.component.html',
    styleUrls: ['./chrono.component.css']
})
export class ChronoComponent implements OnInit {

    constructor() {

    }

    ngOnInit() {

    }
}
```

Puis ajoutez le composant dans la liste des composants de `AppModule` comme suivant :

```
@NgModule({
    declarations: [
        AppComponent,
        ChronoComponent // <- Ici
    ],
    ...
})
export class AppModule { }
```

> Vous devez systématiquement déclarer votre composant en tant que dépendence du module pour l'utiliser !

Exercice 1 - Template

Commencez par créer le template HTML nécessaire à l'affichage des données du chronomètre telles que:

- Les millisecondes,
- Les secondes,
- Les minutes.

Et n'oubliez pas le bouton start/stop du chrono !

Voici un exemple de ce qui est attendu :

Vous pouvez utiliser `@angular/material` pour créer un bouton et votre composant comme suivant :

```
<div>
    <span class="times">0</span>
    <button md-raised-button>Start</button>
</div>
```

Pour le css :

```
:host {
    text-align: center;
    display: block;
    width: 200px;
    height: 200px;
    margin: auto;
    background: #222;
    padding: 30px;
    border-radius: 100%;
    position: relative;
```

```
    z-index: 0;
}
:host>div {
    position: relative;
    z-index: 3;
}
:host:after {
    content: ' ';
    position: absolute;
    width: 206px;
    height: 206px;
    border: 5px solid #414141;
    border-radius: 100%;
    top: -8px;
    left: -8px;
    padding: 30px;
    z-index: 0;
}
:host .times {
    display: block;
    font-size:50px;
    color:white;
    margin-bottom: 50px;
    padding-top: 30px;
}

:hover button {

}
```

> Note : le selecteur `:host` représente l'élément encapsulant votre composant Angular.

Exercice 2 - Création de l'action click

À partir du cours, essayez de créer une action au click du bouton.

Exercice 3

Maintenant que vous avez créé l'action au click du bouton, implémentez les méthodes `startTimer()` et `stopTimer()`.

Notre premier Pipe

En l'état l'affichage des minutes, secondes et millisecondes n'est pas correct (cf. capture chrono).

Nous allons donc créer un Pipe, équivalent des filtres Angular 1, pour formatter les données.

```
src/app
├── app.component.css
├── app.component.html
├── app.component.spec.ts
```

```
├── app.component.ts
├── app.module.ts
├── chrono
│   ├── chrono.component.css
│   ├── chrono.component.html
│   ├── chrono.component.spec.ts
│   └── chrono.component.ts
└── dec-to-str.pipe.ts
```

Commencez par créer le fichier `dec-to-str.pipe.ts`. Créez la `classe DecToStrPipe` comme présenté dans le cours.

Exercice

Le filtre doit transformer/formatter un entier vers un string sur deux chiffres !

> Correction du TP : tp2-solution

TP 3

> Communication entres composants

Dans ce TP, nous allons créer un tableau contenant la liste des utilisateurs avec leur statut en ligne.

Pour rappel, la structure initiale du projet est la suivante :

```
src/app
├── app.component.css
├── app.component.html
├── app.component.spec.ts
├── app.component.ts
└── app.module.ts
```

Le service Users

Nous allons créer notre 1er service. Nous allons partir de principe que l'application va évoluer dans le temps et donc nos services aussi. Par exemple, notre service `UsersService` va exposer un ensemble de méthode dont la liste des utilisateurs. Généralement ces informations sont exposés par un web service. Dans l'immédiat nous n'en avons pas, mais nous ferons tout comme !

Nous allons donc créer deux classes, la première sera un `FakeUsersService` et la seconde sera un `UsersService` (notre cible). Ces deux classes implémenterons l'interface `IUsersService` comme suivant :

```ts
export const Status: {[K in string]: K} = {
    online: "online",
    offline: "offline",
    busy: "busy"
};

export type Status = keyof typeof Status;

export class UserCredential {
    email: string;
    password: string;
}

export class User extends UserCredential {
    id: number;
    firstName: string;
    lastName: string;
    status: Status;
}

export interface IUsersService {
    getUsers(): Promise<User[]>;
    create(user: User): Promise<User>;
    exists(email: string): Promise<boolean>;
    get(email: string): Promise<User>;
}
```

Commencez par créer le UsersService avec @angular/cli :

```
ng g service services/users
```

Ensuite, créez le fichier `users.interface.ts` ainsi que la fichier `users.fake.service.ts`. Et enfin implémentez la méthode `getUsers()` (les autres méthodes seront développés plus tard).

La méthode `FakeUsersService.getUsers()` devra retourner les données suivantes :

```
[
    {"id": 1, "email": "john.doe@gmail.com", "password": "12345", "status": "online"},
    {"id": 2, "email": "jane.doe@gmail.com", "password": "12345", "status": "online"},
    {"id": 3, "email": "jean.dupond@gmail.com", "password": "12345", "status": "busy"},
    {"id": 4, "email": "jean.dupont@gmail.com", "password": "12345", "status": "offline"},
    {"id": 5, "email": "jeanne.dupond@gmail.com", "password": "12345", "status": "offline"},
    {"id": 6, "email": "joe.doe@gmail.com", "password": "12345", "status": "online"}
]
```

Une fois votre service fake implémenté, n'oubliez pas de l'ajouter à la liste des providers de votre module.

> Aidez-vous du cours pour savoir comment ajouter un service mocké dans un module Angular.

Pour terminer, appelez la méthode getUser() dans le composant AppComponent et vérifiez que cela fonctionne.

Le composant tableau des utilisateurs

Nous allons maintenant créer un composant qui va afficher la liste des utilisateurs sous forme de tableau. Commencez par créer le composant `UsersTable` avec la commande suivante :

```
ng g component user-table
```

Voici ce qui est attendu du composant, nous devons pouvoir l'utiliser de la façon suivante :

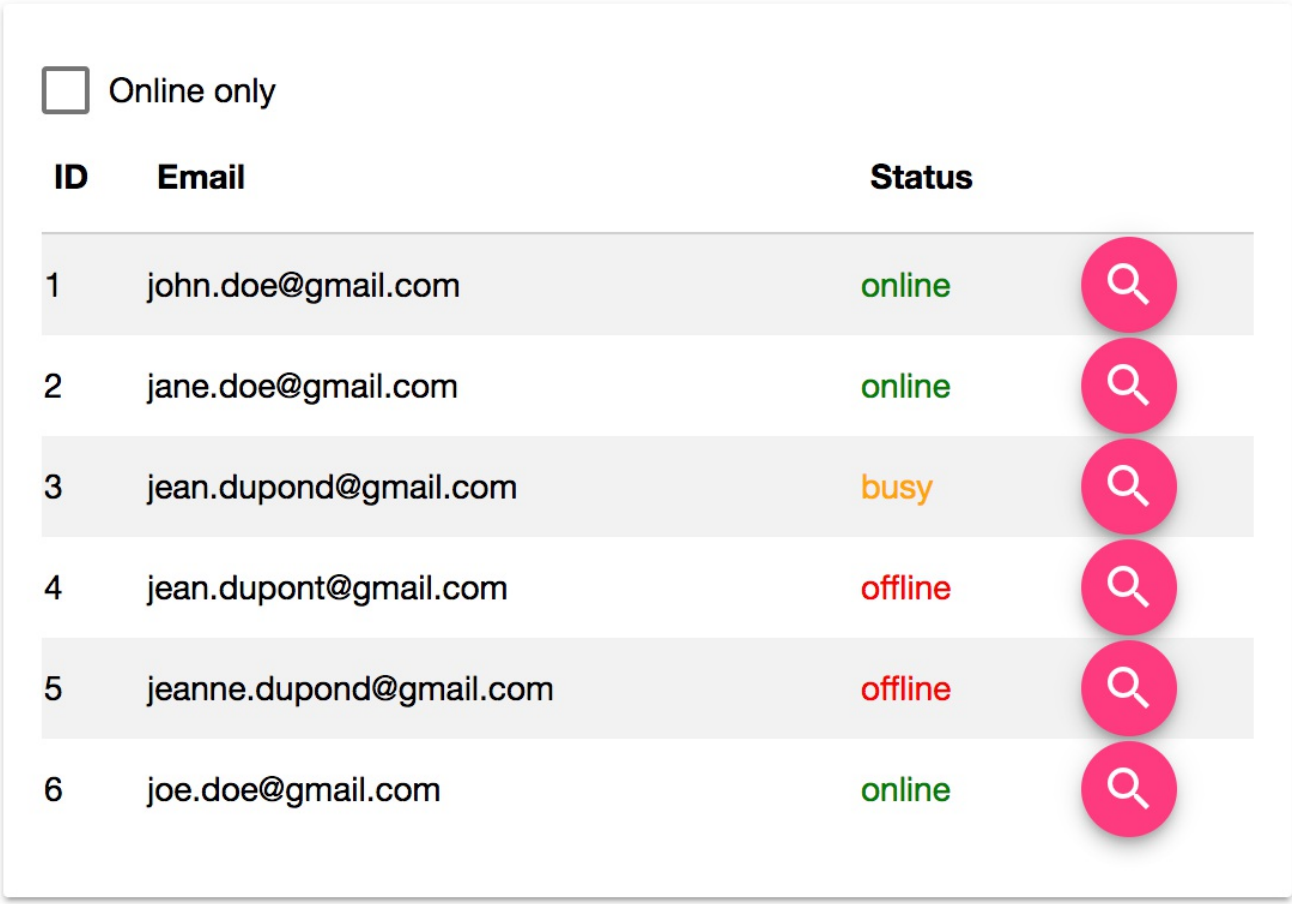
```
<app-users-table [users]="users"></app-users-table>
```

Exercice 1

Avec ce que vous avez vue en cours sur les données d'entrées d'un composant avec `@Input()`, essayez d'afficher la liste des utilisateurs au format json dans le contenu du composant UsersTable dans un premier temps.

Exercice 2

Maintenant que vous avez les données construisez votre composant. Voici un exemple du tableau attendu :



La feuille de style pour vous aider :

```
table {
    border-collapse: collapse;
    border-spacing: 0;
}

table {
    width: 100%;
    display: table;
}

thead {
    border-bottom: 1px solid #d0d0d0;
}

th {
    padding: 15px 5px;
    display: table-cell;
    text-align: left;
    vertical-align: middle;
    border-radius: 2px;
}
```

Vous devez donc compléter la feuille de style afin d'afficher :

- Les lignes pairs en fonds gris clair,

- Le texte `online` en vert,
- Le texte `offline` en gris clair,
- Le texte `busy` en rouge.

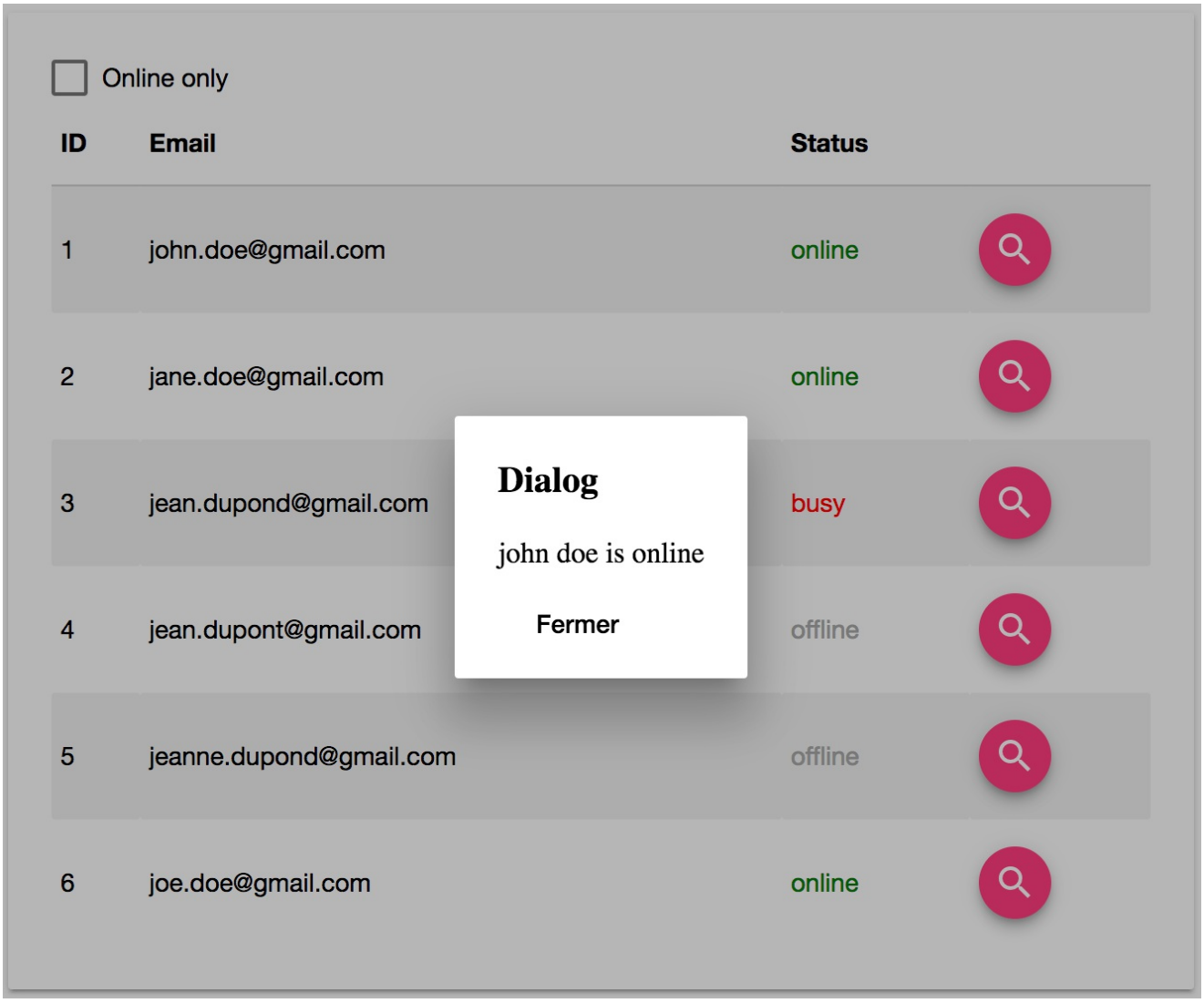
Ensuite utilisez les intructions vue en cours pour afficher la liste des utilisateurs dans votre tableau.

Enfin, ajoutez une checkbox (md-checkbox) permettant d'afficher soit les utilisateurs en ligne, soit tous les utilisateurs.

Communication entres composants

Notre tableau affiche désormais les données. Nous allons maintenant utiliser l'annotation `@Output` pour émettre un événement lorsque l'utilisateur clique sur le bouton correspondant à la ligne de l'utilisateur. Cet événement sera remonté vers le composant `AppComponent`.

Une fois l'information de l'utilisateur remonté dans `AppComponent`, vous devrez afficher les informations de l'utilisateur dans une boite de dialogue (soit via une alert ou en utilisant la librairie Material design).



Conclusion

Ce TP vous aura appris à :

- Utiliser les directives intégrées à Angular (*ngIf,* ngFor),
- Créer un service Angular et l'injecter dans un composant,
- Créer des composants avec une relation parent-enfant,
- Créer et utiliser un template,
- Communiquer entres les composants.

> Correction du TP : tp3-solution

TP 4

> Formulaires et routes

Dans ce TP, nous allons créer les formulaires de connexion et d'authentification. Pour ce faire nous aurons besoin de mettre en place la module de routage d'Angular pour afficher plusieurs pages.

Installation

Installez le module `@angular/router` avec la commande suivante :

```
npm install --save @angular/router
```

Création des pages et des routes

Nous devons maintenant créer les composants Déclarez ensuite les routes dans un nouveau fichier `app.routes.ts`. Nous devrons créer les routes suivantes :

Page	Route	Composant	Description
Accueil	/	HomeComponent	Affiche un message de bienvenue avec un bouton redirigeant l'utilisateur vers la page de connexion. Une fois authentifié, cette page affiche la liste des utilisateurs.
Login	/login	LoginComponent	Formulaire d'authentification
Signup	/signup	SignupComponent	Formulaire d'inscription

1. Commencez par créer les composants énoncés dans le tableau.
2. Ensuite, vous devrez écrire vos routes dans le fichier `app.routes.ts` en fonction de tableau ci-dessus.
3. Importez vos routes dans votre application `app.module.ts` comme suivant :

```
import { ROUTES } from "./app.routes";
import { RouterModule } from "@angular/router";

@NgModule({
    ...
    imports: [
        RouterModule.forRoot(ROUTES)
    ],
    ...
}
```

Enfin vous devez indiquer au routeur où seront affichées les vues dans votre application.

```
<router-outlet></router-outlet>
```

Page d'accueil

La page d'accueil devra afficher la liste des utilisateurs une fois authentifié. Commencez par réintégrer la liste des utilisateurs dans HomeComponent. Nous nous occuperons des règles d'affichage plus tard.

Page d'authentification

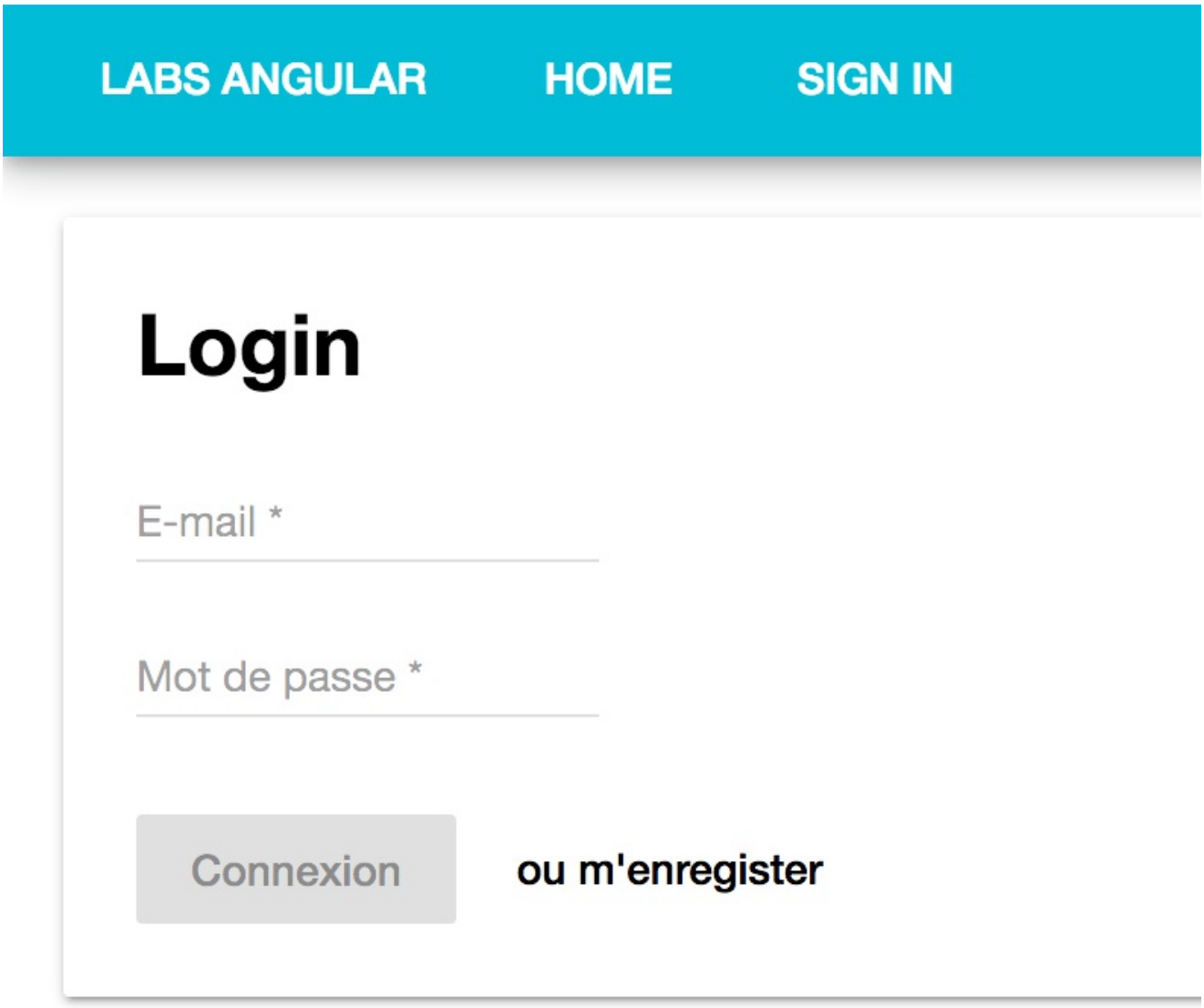
Nous allons maintenant utiliser les API Formulaires d'Angular. Nous utiliserons ici les `formulaires pilotés par le template` mais vous êtes libre d'utiliser la seconde approche.

Le formulaire de connexion présentera les éléments suivants :

- Un champ e-mail, qui est un champ obligatoire et doit respecter le pattern e-mail,
- Un champ mot de passe, qui est un champ obligatoire,
- Un bouton de connexion, actif que si toutes les données saisies sont valides,
- Un bouton `ou m'enregister` pour rediriger l'utilisateur vers la page d'inscription.

N'oubliez pas d'afficher les messages en fonction du type d'erreur du champs associé.

Voici un exemple du formulaire attendu en Material :



Le service Authenticate

Une fois le formulaire réalisé, vous devrez créer le service `AuthenticateService` (`ng g service authenticate`) qui aura pour fonction d'authentifier l'utilisateur.

Ce service exposera les méthodes suivantes :

Method	Descripton
`authenticate(email: string, password: string): boolean`	Permet d'authentifier l'utilisateur. Stocke l'utilisateur dans le `localStorage` et émet un événement sur le flux `onSignin` .
`logout()`	Supprimer les infos stocker dans le `localStorage` et émet un événement sur le flux onLogout.
`getUser()`	Retourne l'utilisateur connecté et stocké dans le localStorage.
`get onSignin(): EventEmitter<User>`	Retourne l'objet (ou flux) `EventEmmiter` dédié à l'événement onSignin.

`get onLogout(): EventEmitter<boolean>`	Retourne l'objet (ou flux) `EventEmmiter` dédié à l'événement onLogout.

> Nous utiliserons la class EventEmitter pour permettre à des composants de s'abonner à des événements.

Utilisation du localStorage

```
// setter
localStorage.setItem("currentUsers", JSON.stringify({firstName: "John"}));
// getter
const user = JSON.parse(localStorage.getItem("currentUsers"));
```

La page d'accueil (fin)

Maintenant que vous avez un service AuthenticateService de prêt, vous pouvez gérer les règles d'affichage de cette page !

Page d'inscription

Le formulaire d'inscription présentera les éléments suivants :

- Prénom, qui est un champ obligatoire,
- Nom, qui est un champ obligatoire,
- Un champ e-mail, qui est un champ obligatoire et doit respecter le pattern e-mail,
- Un champ mot de passe, qui est un champ obligatoire,
- Un bouton de enregistrer, actif que si toutes les données saisies sont valides,

Ensuite rajouter les fonctions permettant d'ajouter un utilisateur dans liste d'utilisateurs du service `UserService`.

La navbar (bonus)

Une application est toujours plus belle lorsque cette dernière possède une `navbar`.

Cette navbar devra s'afficher sur l'ensemble des pages. Elle devra afficher les liens suivants :

- Home,
- Signin lorsque aucun utilisateur n'est authentifié,
- Signout lorsqu'un utilisateur est authentifié.

Pour vous aider, vous pouvez utiliser la librairie Material http://material.angular.io.

> Correction du TP : tp4-solution

TP 5

> Intégration des webservices

Nous avons maintenant besoin de faire fonctionner notre application avec des vrais webservice.

Nous allons donc utiliser le module Http d'Angular pour consommer des services Rest.

Installation du serveur

Le serveur est déjà développé. Il expose un certain nombre de service que nous détaillerons par la suite.

Commençons par l'installer avec la commande suivante :

```
npm install --save labs-angular-backend
npm install --save-dev concurrently
```

Ensuite nous allons créer un fichier `server.js` à la racine du projet. Dans ce fichier copiez le code suivant :

```
const labsAngularBackend = require("labs-angular-backend");

new labsAngularBackend.Server().start().catch(er => console.error(er));
```

Pour nous simplifier la vie nous allons rajouter des commandes npm dans le `package.json :`

```
{
  "scripts": {
      "start": "npm install && concurrently \"npm run start:server\" \"npm run start:app\"",
      "start:app": "ng serve --proxy-config proxy.conf.json",
      "start:server": "node server.js"
  }
}
```

> npm run start démarrera désormais l'application front et le serveur en même temps.

Il nous reste à configurer le proxy de `ng serve` pour que l'application web puisse consommer les webservices. Créez un nouveau fichier `proxy.conf.json` et copiez la configuration suivante :

```
{
```

```
  "/api/*": {
    "target": "http://localhost:8080",
    "changeOrigin": true,
    "secure": false,
    "logLevel": "debug"
  },
  "/socket.io/*": {
    "target": "http://localhost:8080",
    "changeOrigin": true,
    "secure": false,
    "logLevel": "debug"
  }
}
```

Enfin lancez la commande `npm run start` pour vérifier que l'ensemble fonctionne !

Les webservices exposés

Voici la liste des webservices :

Method	Endpoint	Class method
ALL	/api/	RestCtrl.test()
GET	/api/html	RestCtrl.render()
POST	/api/users/	UserCtrl.create()
GET	/api/users/	UserCtrl.getList()
PATCH	/api/users/:email/:status	UserCtrl.updateStatus()
GET	/api/users/:idOrEmail	UserCtrl.get()
PUT	/api/users/:id	UserCtrl.update()
DELETE	/api/users/:id	UserCtrl.remove()
POST	/api/users/authenticate	UserCtrl.authenticate()

Avec les informations dont vous disposez concernant les webservices, vous pouvez integrer ces services dans votre application.

Commencez par integrer le service retournant la liste des utilisateurs ! Ensuite vous pouvez intégrer tous les autres services sauf la suppression (en bonus).

Note : Dans ce TP, vous n'avez pas pour obligation d'utiliser l'API reactive avec le module Http. Vous pouvez directement transformer l'objet Observable retourner par Http en Promise comme suivant :

```
import 'rxjs/add/operator/toPromise';

http.get(`/api/users`).toPromise(); // => Promise
```

> Correction du TP : tp5-solution

TP 6

> Programmation réactive et websocket

Nous allons développer la gestion des changements de statut de connexion des utilisateurs en temps réel.

Autrement dit dès lorsqu'un utilisateur se connectera sur notre application, nous mettrons à jour la liste des utilisateurs avec les nouveaux statuts.

Pour ce faire, nous allons utiliser les WebSockets associés à la programmation réactive.

Installation

Commençons par installer le WebSocket Client:

```
npm install --save socket.io-client
```

Exemple d'utilisation

WebSocket nécessite très de configuration pour fonctionner. Il faut simplement lui donner l'url du serveur websocket. Voici un exemple :

```
import * as io from 'socket.io-client';

const socket = io("http://host:port");

socket.on('mycustomevent', (data) => console.log(data));
```

Exercice 1

Notre objectif sera de mettre en place le socket client et de fournir un observable via votre nouveau service `UsersSocketService`.

Le socket client écoutera l'événement `users.update` et recevra la liste nouvelle liste d'utilisateurs à chaque changement de statuts.

> Pour tester le changement de statuts, ouvrez un nouvelle onglet avec l'application est connectez-vous avec un autre compte utilisateur.

Exercice 2

Une fois que vous avez réussi à récupérer la liste des utilisateurs à l'afficher dans la liste, essayez de rendre observable de bout en bout votre composant. Ainsi nous consommerons à la fois un webservice Ajax en mode Observable et du socket en Observable. Ces observables seront mergés pour ne donner qu'un seul Observable qui sera directement utilisé dans la vue.

Trop difficile ?

Je vous recommande donc de procéder par étape :

- Créer une méthode observable comme par exemple `UserService.getUsersObservable(): Observable<User[]>`,
- Merger les deux observables (UserService et UsersWebSocketService),
- Utiliser le pipe **async** dans le template pour connecter notre Observable avec le `*ngFor`.

À vous de jouer !

> Correction du TP : tp6-solution