Angular

Installation

```
    npm install -g @angular/cli@latest

    ng new projectName --style=scss --directory .
    (--skip-tests=true)

    find . -type f -name '*.spec.ts' -delete
    ng g ngx-spec:specs '**/*'

    ng serve
    ng serve --open
    ng s -o
```

* chrome extensions debugg: augury

_____

Extensions

```
    npm install bootstrap@latest jquery --save
    npm install popper.js --save

    npm i font-awesome --save

    npm install rxjs-compat --save
```

>in angular.json > architect > style ADD:

````ts
    "styles": [
      "../node_modules/bootstrap/dist/css/bootstrap.min.css",
      "src/styles.scss"
    ],

    "scripts": [
      "../node_modules/jquery/dist/jquery.min.js",
      "../node_modules/popper.js/dist/umd/popper.min.js",
      "../node_modules/bootstrap/dist/js/bootstrap.min.js"
    ]
````

````
// OR on style.scss :
@import "~bootstrap/dist/css/bootstrap.css";
````

````html
<script
src="https://use.fontawesome.com/a0a577a7a2.js"></script>
````

> Dans app.component.ts
````ts
import { Component, AfterViewInit } from '@angular/core';


declare var $: any;


export class AppComponent implements  AfterViewInit {

  ngAfterViewInit() {
    // JQUERY
    $(() => {
      $('h2')
      .append($('<span>')
      .css({ color: 'rebeccapurple' })
      .text(' | all OK !'));
    });
  }

}
````

_____

Imports

>in appmodule.ts:
````ts
import { FormsModule } from '@angular/forms';

  imports: [
````

````
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
````

> in new.component.ts -> Décorateur @input()

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

export class newComponent implements OnInit {
  // Propriétés
  @Input() newName: string;
````

Components

```
    ng generate component mon-premier

    ng g c mon-premier
```

Services

```
    ng g s services/auth
```

Models

```
  ng g class models/post --type=model
```

All sub-commands

```
* appShell
* application
* class
```

* component
* directive
* enum
* guard
* interface
* library
* module
* pipe
* service
* serviceWorker
* universal
* webWorker

_____

Usage

* string interpolation :

````html
  <h1>
    Welcome to {{ title }}!
  </h1>
````

* properties binding :

````html
    <button [disabled]="!isAuth">Tout Allumer</button>
````

* events binding :

````html
    <button (click)="onAllumer()">Tout Allumer</button>
````

* liaison à double sens (two-way binding) :

````html
 <input
  type="text"
````

```
  [(ngModel)]="appareilName">
````

> in app.module.ts :
````ts
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    FormsModule
  ]
})
````

* Décorateur @input() :

> dans appareil.component.ts
````ts
import { Component, Input, OnInit } from '@angular/core';

export class AppareilComponent implements OnInit {
  // Propriétés
  @Input() appareilName: string;
  appareilStatus = 'éteint';
````

> dans app.component.html
````html
      <ul class="list-group">
        <!-- by @input() -->
        <app-appareil appareilName="high-tech"></app-appareil>
        <app-appareil appareilName="mon coeur"></app-appareil>
        <app-appareil appareilName="mes
tripes"></app-appareil>
        <!-- by properties blinding -->
        <app-appareil
[appareilName]="appareilOne"></app-appareil>
        <app-appareil
[appareilName]="appareilTwo"></app-appareil>
        <app-appareil
[appareilName]="appareilThree"></app-appareil>
      </ul>
```

````

> dans app.component.ts
```ts
export class AppComponent {
  // propriétés
  appareilOne   = 'Machine à Laver un';
  appareilTwo   = 'Machine à Laver deux';
  appareilThree = 'Machine à Laver trois';
````

_____

Les Directives

Il existe une convention de nomenclature pour les méthodes liées aux événements que j'ai employée ici : "on" + le nom de l'événement. Cela permet, entre autres, de suivre plus facilement l'exécution des méthodes lorsque l'application devient plus complexe.

Directive Structurelle

 * *ngIf : pour afficher des données de façon conditionnelle

    *ngIf="condition"

```html
<li class="list-group-item">
  <div style="width:20px;height:20px;background-color:red;"
       *ngIf="appareilStatus === 'éteint'"></div>
  <h4>Appareil : {{ appareilName }} -- Statut : {{ getStatus()
}}</h4>
  <input type="text" class="form-control"
[(ngModel)]="appareilName">
</li>
````

* *ngFor : pour itérer des données dans un array

    *ngFor="let obj of myArray"
````

````html
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <h2>Mes appareils</h2>
      <ul class="list-group">
        <app-appareil  *ngFor="let appareil of appareils"
                       [appareilName]="appareil.name"

[appareilStatus]="appareil.status"></app-appareil>
      </ul>
      <button class="btn btn-success"
              [disabled]="!isAuth"
              (click)="onAllumer()">Tout allumer</button>
    </div>
  </div>
</div>
````

````ts
export class AppComponent {
  isAuth = false;

  appareils = [
    {
      name: 'Machine à laver',
      status: 'éteint'
    },
    {
      name: 'Frigo',
      status: 'allumé'
    },
    {
      name: 'Ordinateur',
      status: 'éteint'
    }
  ];

  constructor() {
````

Les directives par attribut

* ngStyle

````html
<h4 [ngStyle]="{color: getColor()}">Appareil : {{ appareilName
}} -- Statut : {{ getStatus() }}</h4>
````

````ts
getColor() {
    if(this.appareilStatus === 'allumé') {
      return 'green';
    } else if(this.appareilStatus === 'éteint') {
      return 'red';
    }
}
````

* ngClass

````html
<li [ngClass]="{'list-group-item': true,
                'list-group-item-success': appareilStatus ===
'allumé',
                'list-group-item-danger': appareilStatus ===
'éteint'}">
  <div style="width:20px;height:20px;background-color:red;"
       *ngIf="appareilStatus === 'éteint'"></div>
  <h4 [ngStyle]="{color: getColor()}">Appareil : {{
appareilName }} -- Statut : {{ getStatus() }}</h4>
  <input type="text" class="form-control"
[(ngModel)]="appareilName">
</li>
````

_____

Modifiez les données en temps réel avec les Pipes

Les pipes (/pʌɪp/) prennent des données en input, les transforment, et puis affichent les données modifiées dans le DOM. Il y a des pipes fournis avec Angular, et vous pouvez

également créer vos propres pipes si vous en avez besoin. Je vous propose de commencer avec les pipes fournis avec Angular.

````ts
  lastUpdate = new Promise((resolve, reject) => {
    const date = new Date();
    setTimeout(
      () => {
        resolve(date);
      }, 2000
    );
  });
````

````html
      <p>Mise à jour : {{ lastUpdate | async | date: 'EEEE d MMMM y' | uppercase }}</p>
      <!--  yMMMMEEEEd (E:jour)short -->
````

Les Services

Dit très simplement, un service permet de centraliser des parties de votre code et des données qui sont utilisées par plusieurs parties de votre application ou de manière globale par l'application entière. Les services permettent donc :

* de ne pas avoir le même code doublé ou triplé à différents niveaux de l'application - ça facilite donc la maintenance, la lisibilité et la stabilité du code ;

* de ne pas copier inutilement des données - si tout est centralisé, chaque partie de l'application aura accès aux mêmes informations, évitant beaucoup d'erreurs potentielles.

> créer dossier services/appareil-service.ts
````ts
export class AppareilService {
  appareils = [
    {
````

````
      name: 'Machine à laver',
      status: 'éteint'
    },
    {
      name: 'Frigo',
      status: 'allumé'
    },
    {
      name: 'Ordinateur',
      status: 'éteint'
    }
  ];

  switchOnOne(i: number) { // individuel on
    this.appareils[i].status = 'Allumé!';
  }

  switchOffOne(i: number) { // individuel off
    this.appareils[i].status = 'Eteint!';
  }
}
````

> dans app.module.ts
````ts
import { AppareilService } from './services/appareil-service';

@NgModule({
  providers: [
    AppareilService
  ]
})
````

> dans app.component.ts
````ts
import { Component, OnInit } from '@angular/core';

import { AppareilService } from './services/appareil-service';
````

```
export class AppComponent implements OnInit {

  appareils: any[];

  constructor(
    private appareilService: AppareilService) {
    setTimeout(() => {
      this.isAuth = true;
    }, 4000);
  }

  ngOnInit() {
    this.appareils = this.appareilService.appareils;
  }
````


_____

PS: Faites communiquer les components

````ts
@Input() index: number;
````

````html
<ul class="list-group">
  <app-appareil
  *ngFor="let appareil of appareils; let i = index"

  [index]="i">
  </app-appareil>
</ul>
````

````ts
  onSwitch() {
      this.appareilService.switchOffOne(this.index);
    }
````


_____

Le Routing : navigation

Au lieu de charger une nouvelle page à chaque clic ou à chaque changement d'URL, on remplace le contenu ou une partie du contenu de la page : on modifie les components qui y sont affichés, ou le contenu de ces components. On accomplit tout cela avec le "routing", où l'application lit le contenu de l'URL pour afficher le ou les components requis.

* on vide app-component.html de tous liens components ;
````ts
<app-appareil></app-appareil>
````

* à la place on crée une navigation:
````html
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a href="#">Authentification</a></li>
        <li class="active"><a href="#">Appareils</a></li>
      </ul>
    </div>
  </div>
</nav>
````

Créer des Routes :

> dans app.module.ts

````ts
import { Routes } from '@angular/router';

const appRoutes: Routes = [
  { path: 'appareils', component: AppareilViewComponent },
  { path: 'auth', component: AuthComponent },
  { path: '', component: AppareilViewComponent }
];
// Le path correspond au string qui viendra après le   /  dans l'URL
````

````
imports: [
    RouterModule.forRoot(appRoutes)
],
// méthode  forRoot()  en lui passant l'array de routes que
vous venez de créer
````

* on affiche dans app-component à l'endroit voulu ;
````ts
<router-outlet></router-outlet>
````

Naviguez avec les routerLink

> Pour ne pa recharger la page, on retire l'attribut href et
on le remplace par l'attribut routerLink :

````html
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-collapse">
      <ul class="nav navbar-nav">
        <li routerLinkActive="active">
          <a routerLink="auth">Authentification</a>
        </li>
        <li routerLinkActive="active">
          <a routerLink="appareils">Appareils</a>
        </li>
      </ul>
    </div>
  </div>
</nav>

<div class="container">
      <router-outlet></router-outlet>
</div>
````

Naviguez avec le Router :
Pour naviguer sous condition d'authentification.

> Créez un nouveau fichier auth.service.ts dans le dossier services pour gérer l'authentification :

````ts
export class AuthService {

  isAuth = false;

  signIn() {
    return new Promise(
      (resolve, reject) => {
        setTimeout(
          () => {
            this.isAuth = true;
            resolve(true);
          }, 2000
        );
      }
    );
  }

  signOut() {
    this.isAuth = false;
  }
}
// La variable  isAuth  donne l'état d'authentification de l'utilisateur.  La méthode  signOut()  "déconnecte" l'utilisateur, et la méthode  signIn()  authentifie automatiquement l'utilisateur.
````

> (n'oubliez pas de l'ajouter également dans l'array providers dans app.module.ts)

````ts
  providers: [
    AppareilService,
    AuthService
  ],
````

> Dans le component auth.component.ts , créer deux boutons et les méthodes correspondantes pour se connecter et se

déconnecter contextuellement :

````ts
import { Component, OnInit } from '@angular/core';
import { AuthService } from '../services/auth.service';


export class AuthComponent implements OnInit {

  authStatus: boolean;

  constructor(private authService: AuthService) { }

  ngOnInit() {
    this.authStatus = this.authService.isAuth;
  }

  onSignIn() {
    this.authService.signIn().then(
      () => {
        console.log('Sign in successful!');
        this.authStatus = this.authService.isAuth;
      }
    );
  }

  onSignOut() {
    this.authService.signOut();
    this.authStatus = this.authService.isAuth;
  }

}
// signIn() du service retourne une Promise, on peut employer une fonction callback asynchrone avec  .then()  pour exécuter du code une fois la Promise résolue.
````

> Ajouter les boutons dans auth.component.ts
````html
<h2>Authentification</h2>

<button class="btn btn-success"
````

```
*ngIf="!authStatus"
(click)="onSignIn()">Se connecter</button>

<button class="btn btn-danger"
*ngIf="authStatus"
(click)="onSignOut()">Se déconnecter</button>
````

> Une fois authentifié l'app dirige ver appareil.view.html
Pour ça dans auth.component.ts :
````ts
import { Component, OnInit } from '@angular/core';
import { AuthService } from '../services/auth-service';
import { Router } from '@angular/router';

export class AuthComponent implements OnInit {
  authStatus: boolean;

constructor(private authService: AuthService,
            private router: Router) { }


  onSignIn() {       // Méthode pour les 2 btn connection
    this.authService.signIn().then(
      () => {
        this.authStatus = this.authService.isAuth;
        this.router.navigate(['appareils']);
        // Auth OK! -> fonction navigate dirige vers le chemin
'appareils'
      }
    )
  }
````

Paramètres de routes
Imaginez qu'on souhaite pouvoir cliquer sur un appareil dans la liste d'appareils afin d'afficher une page avec plus d'informations sur cet appareil : on peut imaginer un système de routing de type appareils/nom-de-l'appareil , par exemple.
 Si on n'avait que deux ou trois appareils, on pourrait être tenté de créer une route par appareil, mais imaginez un cas de figure où l'on aurait 30 appareils, ou 300. Imaginez qu'on

laisse l'utilisateur créer de nouveaux appareils ; l'approche de créer une route par appareil n'est pas adaptée. Dans ce genre de cas, on choisira plutôt de créer une route avec paramètre.

> Tout d'abord, vous allez créer la route dans AppModule :
````ts
const appRoutes: Routes = [
  { path: 'appareils', component: AppareilViewComponent },
  { path: 'appareils/:id', component: SingleAppareilComponent
},
];
````
> Tous les chemins de type appareils/* seront renvoyés vers SingleAppareilComponent , que vous allez maintenant créer :
````ts
import { Component, OnInit } from '@angular/core';
import { AppareilService } from
'../services/appareil.service';


export class SingleAppareilComponent implements OnInit {

  name: string = 'Appareil';
  status: string = 'Statut';

  constructor(private appareilService: AppareilService) { }

  ngOnInit() {
  }

}
````


````html
<h2>{{ name }}</h2>
<p>Statut : {{ status }}</p>
<a routerLink="/appareils">Retour à la liste</a>
````
Pour l'instant, si vous naviguez vers /appareils/nom , peu importe le nom que vous choisissez, vous avez accès à SingleAppareilComponent .

>Maintenant, vous allez y injecter ActivatedRoute , importé depuis @angular/router , afin de récupérer le fragment id de l'URL :
````ts
import { AppareilService } from
'../services/appareil-service';
import { ActivatedRoute } from '@angular/router';


constructor(private appareilService: AppareilService,
            private route: ActivatedRoute) { }

ngOnInit() {
    this.name = this.route.snapshot.params['id'];
}
// utiliser l'objet  snapshot  qui contient les paramètres de l'URL  et, pour l'instant, attribuer le paramètre  id  à la variable  name
````
> dans AppareilService , ID pour chaque appareil et une méthode qui rendra l'appareil correspondant à un identifiant :
````ts
appareils = [
    {
      id: 1,
      name: 'Machine à laver',
      status: 'éteint'
    },
];


  getAppareilById(id: number) {
    const appareil = this.appareils.find(
      (appareilObject) => {
        return appareilObject.id === id;
      }
    );
    return appareil;
  }                   // méthodes pour ID:
````

> Dans SingleAppareilComponent, récupérer l'identifiant de

l'URL et l'utiliser pour récupérer l'appareil correspondant :
````ts
ngOnInit() {
    const id = this.route.snapshot.params['id'];

    this.name =
this.appareilService.getAppareilById(+id).name;
    this.status =
this.appareilService.getAppareilById(+id).status;
}
// utiliser  +  avant  id  dans l'appel pour caster la
variable comme nombre.
````
> ne pas oublier dans AppareilComponent :
````ts
@Input() appareilName: string;
@Input() appareilStatus: string;
@Input() index: number;
@Input() id: number;
````
> et dans AppareilViewComponent
````html
<ul class="list-group">
  <app-appareil  *ngFor="let appareil of appareils; let i =
index"
                 [appareilName]="appareil.name"
                 [appareilStatus]="appareil.status"
                 [index]="i"
                 [id]="appareil.id"></app-appareil>
</ul>
````
> dans SingleAppareil
````html
<h3>{{ name }}</h3>
<p>Status : {{ status }}</p>
<a routerLink="/appareils">Retour à la liste</a>
````
> dans AppareilComponent
````html
<h4 [ngStyle]="{color: getColor()}">
  Appareil : {{ appareilName }} --
  Statut : {{ getStatus() }}</h4>
````

```
<a [routerLink]="[id]">Détail</a>

<!-- Ici, vous utilisez le format array pour  routerLink  en property binding afin d'accéder à la variable  id -->
```
Accès à la page Détail pour chaque appareil, informations de statut qui s'y trouvent sont automatiquement à jour grâce à l'utilisation du service.

Redirection 404 :
Par exemple pour afficher une page 404 lorsqu'il entre une URL qui n'existe pas.

> créer un component 404 très simple, appelé four-oh-four.component.ts :
```html
<h2>Erreur 404</h2>
<p>La page que vous cherchez n'existe pas !</p>

```
> Ajouter la route "directe" vers cette page, ainsi qu'une route "wildcard", qui redirigera toute route inconnue vers la page d'erreur :
```ts
const appRoutes: Routes = [

  { path: 'not-found', component: FourOhFourComponent },
  { path: '**', redirectTo: 'not-found' }

];
// ous êtes redirigé vers  /not-found  et donc le component 404.
```

Les Guards
Il peut y avoir des cas de figure où vous souhaiterez exécuter du code avant qu'un utilisateur puisse accéder à une route ; par exemple, vous pouvez souhaiter vérifier son authentification ou son identité. Techniquement, ce serait possible en insérant du code dans la méthode ngOnInit() de chaque component, mais cela deviendrait lourd, avec du code

répété et une multiplication des erreurs potentielles. Ce serait donc mieux d'avoir une façon de centraliser ce genre de fonctionnalité. Pour cela, il existe la guard canActivate .

Une guard est en effet un service qu'Angular exécutera au moment où l'utilisateur essaye de naviguer vers la route sélectionnée. Ce service implémente l'interface canActivate , et donc doit contenir une méthode du même nom qui prend les arguments ActivatedRouteSnapshot et RouterStateSnapshot (qui lui seront fournis par Angular au moment de l'exécution) et retourne une valeur booléenne, soit de manière synchrone (boolean), soit de manière asynchrone (sous forme de Promise ou d'Observable) :

> Dans services, auth-guard.service.ts :

````ts
import {
ActivatedRouteSnapshot,
CanActivate,
Router,
RouterStateSnapshot } from "@angular/router";
import { Observable } from "rxjs/Observable";

import { AuthService } from "./auth-service";
import { Injectable } from '@angular/core';


// Injecter un service à travers 1 autre service
@Injectable()


export class AuthGuard implements CanActivate{

        constructor(private authService: AuthService,

private router: Router) { }

        canActivate(
                route: ActivatedRouteSnapshot,
                state: RouterStateSnapshot
        ): Observable<boolean> | Promise<boolean> | boolean {
````

```
                if (this.authService.isAuth) {
                        return true;
                } else {
                        this.router.navigate(['/auth']);
                }

        }
}
// À l'intérieur de la méthode  canActivate() , vous allez vérifier l'état de l'authentification dans  AuthService .  Si l'utilisateur est authentifié, la méthode renverra  true , permettant l'accès à la route protégée.  Sinon, vous pourriez retourner  false , mais cela empêchera simplement l'accès sans autre fonctionnalité. Rediriger l'utilisateur vers la page d'authentification, le poussant à s'identifier
````

> Pour appliquer cette garde à la route /appareils et à toutes ses routes enfants, il faut l'ajouter dans AppModule . N'oubliez pas d'ajouter AuthGuard à l'array providers , puisqu'il s'agit d'un service :

````ts
const appRoutes: Routes = [
  // canActive pour guarder l'accès appareils

  { path: 'appareils',
  canActivate: [AuthGuard],
  component: AppareilViewComponent },

  { path: 'appareils/:id',
  canActivate: [AuthGuard],
  component: SingleAppareilComponent },

  { path: 'appareil',
  component: AppareilComponent },
  { path: 'auth',
  component: AuthComponent },
  { path: '',
  component: AppareilViewComponent },
  { path: 'no-found',
  component: QuatreCentQuatreComponent },
```

```
  // Page 404
  { path: '**',
  redirectTo: '/not-found'}
]


  providers: [
    AppareilService,
    AuthService,
    AuthGuard
  ],
````

* gérer le routing de votre application avec des routerLink et de manière programmatique (avec router.navigate())
* comment rediriger un utilisateur
* protéger des routes de votre application avec les guards.

_____
Observez les données avec RXJS

Pour réagir à des événements ou à des données de manière asynchrone (c'est-à-dire ne pas devoir attendre qu'une tâche, par exemple un appel HTTP, soit terminée avant de passer à la ligne de code suivante), il y a eu plusieurs méthodes depuis quelques années. Il y a le système de callback, par exemple, ou encore les Promise. Avec l'API RxJS, fourni et très intégré dans Angular, la méthode proposée est celle des Observables

> npm i rxjs-compat
Observables

Observable est un objet qui émet des informations auxquelles on souhaite réagir. Ces informations peuvent venir d'un champ de texte dans lequel l'utilisateur rentre des données, ou de la progression d'un chargement de fichier, Elles peuvent également venir de la communication avec un serveur : le client HTTP.
> Dans appComponent:
````ts
````

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/interval';


export class AppComponent implements OnInit {

secondes: number;

ngOnInit() {
    const counter = Observable.interval(1000);

    counter.subscribe(
      (value) => {
        this.secondes = value;
      },
      (error) => {
        console.log('Uh-oh, an error occurred! : ' + error);
      },
      () => {
        console.log('Observable complete!');
      }
    );
}
// À cet Observable, on associe un Observer – un bloc de code qui sera exécuté à chaque fois que l'Observable émet une information.  L'Observable émet trois types d'information : des données, une erreur, ou un message complete.  Du coup, tout Observer peut avoir trois fonctions : une pour réagir à chaque type d'information.
````
> Dans appComponent.html:
````html
        <span class="badge badge-primary ml-3">Time:  {{ secondes }}s</span>
````

Subscriptions
Dans le chapitre précédent, vous avez appris à créer un Observable et à vous y souscrire. Pour rappel, la fonction subscribe() prend comme arguments trois fonctions anonymes :

* la première se déclenche à chaque fois que l'Observable émet de nouvelles données, et reçoit ces données comme argument ;

* la deuxième se déclenche si l'Observable émet une erreur, et reçoit cette erreur comme argument ;

* la troisième se déclenche si l'Observable s'achève, et ne reçoit pas d'argument.

> Afin d'éviter tout problème, quand vous utilisez des Observables personnalisés, il est vivement conseillé de stocker la souscription dans un objet Subscription (à importer depuis rxjs/Subscription) :

````ts
import { Component, OnInit, OnDestroy} from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/interval';
import { Subscription } from 'rxjs/Subscription';

export class AppComponent implements OnInit {

  secondes: number;
  counterSubscription: Subscription;

  ngOnInit() {
    const counter = Observable.interval(1000);

    this.counterSubscription = counter.subscribe(
      (value) => {      // 1
        this.secondes = value;
      },
      (error) => {     // 2
        console.log('Uh-oh, an error occurred! : ' + error);
      },
      () => {          // 3
        console.log('Observable complete!');
      }
    );

    ngOnDestroy() {
      this.counterSubscription.unsubscribe();
````

```
    }

  }
}

// La fonction  unsubscribe()  détruit la souscription et
empêche les comportements inattendus liés aux Observables
infinis, donc n'oubliez pas de unsubscribe ! après avoir
subscribe !
````

Subjects ! (problèmes de versions) !

Il existe un type d'Observable qui permet non seulement de réagir à de nouvelles informations, mais également d'en émettre. Imaginez une variable dans un service, par exemple, qui peut être modifié depuis plusieurs components ET qui fera réagir tous les components qui y sont liés en même temps. Voici l'intérêt des Subjects.

* rendre l'array des appareils private ;

* créer un Subject dans le service ;

* créer une méthode qui, quand le service reçoit de nouvelles données, fait émettre ces données par le Subject et appeler cette méthode dans toutes les méthodes qui en ont besoin ;

* souscrire à ce Subject depuis AppareilViewComponent pour recevoir les données émises, émettre les premières données, et implémenter OnDestroy pour détruire la souscription.
> dans AppService:
````ts
import { Subject } from 'rxjs/Subject';

export class AppareilService {

  appareilsSubject = new Subject<any[]>();

  private appareils = [
    {
      id: 1,
```

```
      name: 'Machine à laver',
      status: 'éteint'
    },
    // ... 2 et 3
  ];

emitAppareilSubject() {
    this.appareilsSubject.next(this.appareils.slice());
  }

switchOnAll() {
    for(let appareil of this.appareils) {
      appareil.status = 'allumé';
    }
    this.emitAppareilSubject();
}

switchOffAll() {
    for(let appareil of this.appareils) {
      appareil.status = 'éteint';
      this.emitAppareilSubject();
    }
}

switchOnOne(i: number) {
    this.appareils[i].status = 'allumé';
    this.emitAppareilSubject();
}

switchOffOne(i: number) {
    this.appareils[i].status = 'éteint';
    this.emitAppareilSubject();
}
// Quand vous déclarez un Subject, il faut dire quel type de
données il gèrera.  Puisque nous n'avons pas créé d'interface
pour les appareils (vous pouvez le faire si vous le
souhaitez), il gèrera des array de type  any[] .  N'oubliez
pas l'import !
````


> Dans AppareilViewComponent
````ts
```

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { AppareilService } from
'../services/appareil.service';
import { Subscription } from 'rxjs/Subscription';

@Component({
  selector: 'app-appareil-view',
  templateUrl: './appareil-view.component.html',
  styleUrls: ['./appareil-view.component.scss']
})
export class AppareilViewComponent implements OnInit,
OnDestroy {

  appareils: any[];
  appareilSubscription: Subscription;

  lastUpdate = new Promise((resolve, reject) => {
    const date = new Date();
    setTimeout(
      () => {
        resolve(date);
      }, 2000
    );
  });

  constructor(private appareilService: AppareilService) { }

  ngOnInit() {
    this.appareilSubscription =
this.appareilService.appareilsSubject.subscribe(
      (appareils: any[]) => {
        this.appareils = appareils;
      }
    );
    this.appareilService.emitAppareilSubject();
  }

  onAllumer() {
    this.appareilService.switchOnAll();
  }

  onEteindre() {
```

```
    if(confirm('Etes-vous sûr de vouloir éteindre tous vos
appareils ?')) {
      this.appareilService.switchOffAll();
    } else {
      return null;
    }
  }

  ngOnDestroy() {
    this.appareilSubscription.unsubscribe();
  }

}
// L'application refonctionne comme avant, mais avec une
différence cruciale de méthodologie : il y a une abstraction
entre le service et les components, où les données sont
maintenues à jour grâce au Subject.
````


Pourquoi est-ce important ?

Dans ce cas précis, ce n'est pas fondamentalement nécessaire, mais imaginez qu'on intègre un système qui vérifie périodiquement le statut des appareils. Si les données sont mises à jour par une autre partie de l'application, il faut que l'utilisateur voie ce changement sans avoir à recharger la page. Il va de même dans l'autre sens : un changement au niveau du view doit pouvoir être reflété par le reste de l'application sans rechargement.

Opérateurs

L'API RxJS propose énormément de possibilités:
* [Documentation GIT RxJS](https://github.com/ReactiveX/rxjs)
* [Get Started Guide](https://rxjs.dev/guide/overview)
* [API Dev List](https://rxjs.dev/api)

Comme les opérateurs:

Un opérateur est une fonction qui se place entre l'Observable et l'Observer (la Subscription, par exemple), et qui peut filtrer et/ou modifier les données reçues avant même qu'elles

n'arrivent à la Subscription.
Voici quelques exemples rapides :

* map() : modifie les valeurs reçues – peut effectuer des calculs sur des chiffres, transformer du texte, créer des objets…

* filter() : comme son nom l'indique, filtre les valeurs reçues selon la fonction qu'on lui passe en argument.

* throttleTime() : impose un délai minimum entre deux valeurs – par exemple, si un Observable émet cinq valeurs par seconde, mais ce sont uniquement les valeurs reçues toutes les secondes qui vous intéressent, vous pouvez passer throttleTime(1000) comme opérateur.

* scan() et reduce() : permettent d'exécuter une fonction qui réunit l'ensemble des valeurs reçues selon une fonction que vous lui passez – par exemple, vous pouvez faire la somme de toutes les valeurs reçues. La différence basique entre les deux opérateurs : reduce() vous retourne uniquement la valeur finale, alors que scan() retourne chaque étape du calcul.

_____
Écoutez l'utilisateur avec les Forms - METHODE TEMPLATE
En Angular, il y a deux grandes méthodes pour créer des formulaires :

* la méthode template : vous créez votre formulaire dans le template, et Angular l'analyse pour comprendre les différents inputs et pour en mettre à disposition le contenu ;

Créer un formulaire et Validez les données

> Créer EditAppareilComponent:
````html
<form (ngSubmit)="onSubmit(f)" #f="ngForm">

  <div class="form-group">
      <label for="name">
        Nom de l'appareil
````

```
      </label>
      <input type="text" id="name" class="form-control"
name="name" ngModel required>
  </div>
  <div class="form-group">
      <label for="status">
        État de l'appareil
      </label>
      <select id="status" class="form-control" name="status"
[ngModel]="defaultOnOff">
            <option value="allumé">Allumé</option>
            <option value="éteint">Éteint</option>
      </select>
  </div>

  <button
  class="btn btn-primary"
  type="submit"
  [disabled]="f.invalid">Enregistrer</button>

</form>
<!-- L'attribut  #f  est ce qu'on appelle une référence
locale.  Vous donnez simplement un nom à l'objet sur lequel
vous ajoutez cet attribut ;  -->
````


> Dans edit-appareil.component.ts:
````ts

export class EditAppareilComponent implements OnInit {

// variable TypeScript pour choix select
  defaultOnOff = 'éteint';

  constructor() { }

// méthode pour submit form:

  onSubmit(form: NgForm) {
    const name = form.value['name'];
    const status = form.value['status'];
```

```
    // here methode create on AppareilService
    this.appareilService.addAppareil(name, status);
    console.log(form.value);

    this.router.navigate(['/appareils']);
  }

// Ici vous utilisez la propriété  value  du  NgForm . L'objet
 NgForm  (à importer depuis  @angular/forms )
````


* [voir doc officielle
ngForm]('https://angular.io/api/forms/NgForm')

> On rajoute path dans AppModule.ts
````ts
const appRoutes: Routes = [
  { path: 'appareils', canActivate: [AuthGuard], component:
AppareilViewComponent },

  { path: 'edit', canActivate: [AuthGuard], component:
EditAppareilComponent },
];
````
> ainsi dans Nav AppComponent
````html
<ul class="nav navbar-nav">
    <li routerLinkActive="active"><a
routerLink="auth">Authentification</a></li>
    <li routerLinkActive="active"><a
routerLink="appareils">Appareils</a></li>
    <li routerLinkActive="active"><a routerLink="edit">Nouvel
appareil</a></li>
</ul>
````

Exploitez les données

> Dans la méthode onSubmit() , vous allez récupérer les
données et les attribuer à deux constantes pour les envoyer à
AppareilService via une méthode :

````ts
// Méthode:

addAppareil(name: string, status: string) {
    const appareilObject = {
      id: 0,
      name: '',
      status: ''
    };
    appareilObject.name = name;
    appareilObject.status = status;
    appareilObject.id = this.appareils[(this.appareils.length
- 1)].id + 1;
    this.appareils.push(appareilObject);
    this.emitAppareilSubject();
}
// Cette méthode crée un objet du bon format, et attribue le
nom et le statut qui lui sont passés comme arguments.  La
ligne pour l'id prend l'id du dernier élément actuel de
l'array et ajoute 1.  Ensuite, l'objet complété est ajouté à
l'array et le Subject est déclenché pour tout garder à jour.
````

Écoutez l'utilisateur avec les Forms - METHODE REACTIVE

* la méthode réactive : vous créez votre formulaire en TypeScript et dans le template, puis vous en faites la liaison manuellement – si cela a l'air plus complexe, cela vous permet de gérer votre formulaire en détail, notamment avec la création programmatique de contrôles (permettant, par exemple, à l'utilisateur d'ajouter des champs).

> Dans User.model.ts:
````ts
export class User{

        constructor(
                public firstName: string,
                public lastName: string,
                public email: string,
                public drinkPreference: string,
                public hobbies?: string[]) {
````

```
                }
}
//  Ce modèle pourra donc être utilisé dans le reste de
l'application en l'important dans les components où vous en
avez besoin.  Cette syntaxe de constructeur permet
l'utilisation du mot-clé  new , et les arguments passés seront
attribués à des variables qui portent les noms choisis ici,
par exemple  const user = new User('James', 'Smith',
'james@james.com', 'jus d\'orange', ['football', 'lecture'])
créera un nouvel objet  User  avec ces valeurs attribuées aux
variables  user.firstName ,  user.lastName  etc.
````


> Ensuite, créez un UserService simple qui stockera la liste
des objets User et qui comportera une méthode permettant
d'ajouter un utilisateur à la liste :
````ts
import { User } from '../models/User.model';
import { Subject } from 'rxjs/Subject';

export class UserService {
  private users: User[];
  userSubject = new Subject<User[]>();

  emitUsers() {
    this.userSubject.next(this.users.slice());
  }

  addUser(user: User) {
    this.users.push(user);
    this.emitUsers();
  }
}
// Ce service contient un array privé d'objets de type
personnalisé  User  et un Subject pour émettre cet array.  La
méthode  emitUsers()  déclenche ce Subject et la méthode
addUser()  ajoute un objet  User  à l'array, puis déclenche le
Subject.
````


* N'oubliez pas d'ajouter ce nouveau service à l'array
providers dans AppModule !

> L'étape suivante est de créer UserListComponent – pour simplifier cet exemple, vous n'allez pas créer un component pour les utilisateurs individuels :

````ts
import { Component, OnDestroy, OnInit } from '@angular/core';
import { User } from '../models/User.model';
import { Subscription } from 'rxjs/Subscription';
import { UserService } from '../services/user.service';


export class UserListComponent implements OnInit, OnDestroy {

  users: User[];
  userSubscription: Subscription;

  constructor(private userService: UserService) { }

  ngOnInit() {
    this.userSubscription =
this.userService.userSubject.subscribe(
      (users: User[]) => {
        this.users = users;
      }
    );
    this.userService.emitUsers();
  }



  ngOnDestroy() {
    this.userSubscription.unsubscribe();
  }

}
// Ce component très simple souscrit au Subject dans
UserService  et le déclenche pour en récupérer les
informations et les rendre disponibles au template (que vous
allez maintenant créer) :
````

> Dans user-list.component.html:

````html
<ul class="list-group">
  <li class="list-group-item" *ngFor="let user of users">
    <h3>{{ user.firstName }} {{ user.lastName }}</h3>
    <p>{{ user.email }}</p>
    <p>Cette persone préfère le {{ user.drinkPreference }}</p>
    <p *ngIf="user.hobbies && user.hobbies.length > 0">
      Cette personne a des hobbies !
      <span *ngFor="let hobby of user.hobbies">{{ hobby }} -
</span>
    </p>
  </li>
</ul>

<!-- Ici, vous appliquez des directives  *ngFor  et  *ngIf
pour afficher la liste des utilisateurs et leurs hobbies,
s'ils en ont. -->
````

* N'oubliez pas d'ajouter le lien "Users" dans NAV et son PATH !

> Ajoutez également un objet User codé en dur dans le service pour voir les résultats :

````ts
        private users: User[] = [
                {
                        firstName: 'Gilius',
                        lastName: 'Blanchard',
                        email: 'neworldwebsites@gmail.com',
                        drinkPreference: 'fanta',
                        hobbies: [
                                'coding',
                                'Dégustation de café'   ]
                }
        ];
````

Construisez un formulaire avec FormBuilder

Dans la méthode template, l'objet formulaire mis à disposition par Angular était de type `NgForm`, mais ce n'est pas le cas pour les formulaires réactifs. Un formulaire réactif est de type `FormGroup`, et il regroupe plusieurs `FormControl` (tous les deux importés depuis `@angular/forms`).

> Vous commencez d'abord, donc, par créer l'objet dans votre nouveau component `NewUserComponent` :

````ts
// Ensuite, vous allez créer une méthode qui sera appelée dans le constructeur pour la population de cet objet, et vous allez également injecter `FormBuilder`, importé depuis `@angular/forms`

import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators, FormArray } from '@angular/forms';
import { UserService } from '../services/user.service';
import { Router } from '@angular/router';
import { User } from '../models/User.model';


export class NewUserComponent implements OnInit {

  userForm: FormGroup;

  constructor(private formBuilder: FormBuilder,
              private userService: UserService,
              private router: Router) { }

  ngOnInit() {
    this.initForm();
  }

// FormBuilder est une classe qui vous met à disposition des méthodes facilitant la création d'objet `FormGroup`. Vous allez maintenant utiliser la méthode `group` à l'intérieur de votre méthode `initForm()` pour commencer à créer le formulaire :
````

```
initForm() {
    this.userForm = this.formBuilder.group({
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
      drinkPreference: ['', Validators.required]
    });
}
```

// La méthode `group` prend comme argument un objet où les clés correspondent aux noms des contrôles souhaités et les valeurs correspondent aux valeurs par défaut de ces champs. Puisque l'objectif est d'avoir des champs vides au départ, chaque valeur ici correspond au string vide.

```
  onSubmitForm() {
    const formValue = this.userForm.value;
    const newUser = new User(
      formValue['firstName'],
      formValue['lastName'],
      formValue['email'],
      formValue['drinkPreference']
    );
    this.userService.addUser(newUser);
    this.router.navigate(['/users']);
  }
```

// La méthode `onSubmitForm()` récupère la `value` du formulaire, et crée un nouvel objet `User` (à importer en haut) à partir de la valeur des `controls` du formulaire. Ensuite, elle ajoute le nouvel utilisateur au service et navigue vers `/users` pour en montrer le résultat.

```
  getHobbies() {
    return this.userForm.get('hobbies') as FormArray;
  }
```

// Afin d'avoir accès aux `controls` à l'intérieur de l'array, pour des raisons de typage strict liées à TypeScript, il faut créer une méthode qui retourne `hobbies` par la méthode `get()` sous forme de `FormArray` (`FormArray` s'importe depuis `@angular/forms`)

```
  // Ensuite, vous allez créer la méthode qui permet d'ajouter un FormControl à hobbies , permettant ainsi à l'utilisateur d'en ajouter autant qu'il veut. Vous allez également rendre le nouveau champ requis, afin de ne pas avoir un array de hobbies avec des string vides !
  onAddHobby() {
    const newHobbyControl = this.formBuilder.control('', Validators.required);
    this.getHobbies().push(newHobbyControl);
  }
  // Cette méthode crée un control avec la méthode FormBuilder.control() , et l'ajoute au FormArray rendu disponible par la méthode getHobbies() .

}
````

> Dans new-user.component.html créer le template du formulaire et lier ce template à l'objet userForm que vous venez de créer :

````html
  <form [formGroup]="userForm" (ngSubmit)="onSubmitForm()">

    <div class="form-group">
      <label for="firstName">Prénom</label>
      <input type="text" id="firstName" class="form-control" formControlName="firstName">
    </div>
    <div class="form-group">
      <label for="lastName">Nom</label>
      <input type="text" id="lastName" class="form-control" formControlName="lastName">
    </div>
    <div class="form-group">
      <label for="email">Adresse e-mail</label>
      <input type="text" id="email" class="form-control" formControlName="email">
    </div>
    <div class="form-group">
```

```
      <label for="drinkPreference">Quelle boisson
préférez-vous ?</label>
      <select id="drinkPreference" class="form-control"
formControlName="drinkPreference">
        <option value="jus d\'orange">Jus d'orange</option>
        <option value="jus de mangue">Jus de mangue</option>
      </select>
    </div>

            <!-- Hobbies optionnel array  -->
    <div formArrayName="hobbies">
          <h3>Vos hobbies</h3>
          <div class="form-group" *ngFor="let hobbyControl of
getHobbies().controls; let i = index">
            <input type="text" class="form-control"
[formControlName]="i">
          </div>
          <button type="button" class="btn btn-success"
(click)="onAddHobby()">Ajouter un hobby</button>
    </div>
    <button type="submit" class="btn btn-primary"
[disabled]="userForm.invalid">Soumettre</button>


          <!-- SUBMIT -->
    <button type="submit" class="btn
btn-primary">Soumettre</button>

  </form>
```
Analysez le template :

* Sur la balise  FORM , vous utilisez le property binding pour lier l'objet  userForm  à l'attribut  formGroup  du formulaire, créant la liaison pour Angular entre le template et le TypeScript.

* Également dans la balise  FORM , vous avez toujours une méthode  onSubmitForm()  liée à ngSubmit, mais vous n'avez plus besoin de passer le formulaire comme argument puisque vous y avez déjà accès par l'objet  userForm  que vous avez créé.

* Sur chaque `INPUT` qui correspond à un `control` du formulaire, vous ajoutez l'attribut `formControlName` où vous passez un string correspondant au nom du `control` dans l'objet TypeScript.

* Le bouton de type `submit` déclenche l'événement `ngSubmit`, déclenchant ainsi la méthode `onSubmitForm()`, que vous allez créer dans votre TypeScript.

> Il ne reste plus qu'à ajouter un lien dans `UserListComponent` qui permet d'accéder à `NewUserComponent` et de créer la route correspondante `new-user` dans `AppModule`

````html
<ul class="list-group">
  <li class="list-group-item" *ngFor="let user of users">
    <h3>{{ user.firstName }} {{ user.lastName }}</h3>
    <p>{{ user.email }}</p>
    <p>Cette persone préfère le {{ user.drinkPreference }}</p>
    <p *ngIf="user.hobbies && user.hobbies.length > 0">
      Cette personne a des hobbies !
      <span *ngFor="let hobby of user.hobbies">{{ hobby }} -
</span>
    </p>
  </li>

  <a routerLink="/new-user">Nouvel utilisateur</a>
  <!-- Puisque ce `routerLink` se trouve à l'intérieur du router-outlet, il faut ajouter un `/` au début de l'URL pour naviguer vers `localhost:4200/new-user`. Si vous ne mettez pas le `/`, ce lien naviguera vers localhost:4200/users/new-user -->

</ul>
````

````ts
{ path: 'users', component: UserListComponent },
{ path: 'new-user', component: NewUserComponent },
````

Validators

Comme pour la méthode template, il existe un outil pour la validation de données dans la méthode réactive : les Validators

> Pour ajouter la validation, vous allez modifier légèrement votre exécution de FormBuilder.group VOIR DANS new-user.component

 * [Voir Documentation Validators](https://angular.io/guide/form-validation#validator-functions)

Ajoutez dynamiquement des FormControl
Pour l'instant, vous n'avez pas encore laissé la possibilité à l'utilisateur d'ajouter ses hobbies. Il serait intéressant de lui laisser la possibilité d'en ajouter autant qu'il veut, et pour cela, vous allez utiliser un FormArray . Un FormArray est un array de plusieurs FormControl , et permet, par exemple, d'ajouter des nouveaux controls à un formulaire. Vous allez utiliser cette méthode pour permettre à l'utilisateur d'ajouter ses hobbies.

> Modifiez d'abord initForm() pour ajouter un FormArray vide qui s'appellera hobbies avec la méthode array : VOIR new-user.component.ts

Analysez cette DIV :

````html
<div formArrayName="hobbies">
      <h3>Vos hobbies</h3>

      <div class="form-group"
      *ngFor="let hobbyControl of getHobbies().controls;
      let i = index">

          <input type="text" class="form-control"
          [formControlName]="i">
````

```
      </div>

      <button type="button"
      class="btn btn-success"
      (click)="onAddHobby()">
      Ajouter un hobby</button>
</div>

<button
type="submit"
class="btn btn-primary"
[disabled]="userForm.invalid">
Soumettre</button>
<!-- En liant la validité de  userForm  à la propriété
disabled  du bouton  submit , vous intégrez la validation de
données : -->
````

* à la DIV qui englobe toute la partie hobbies , vous ajoutez l'attribut formArrayName , qui correspond au nom choisi dans votre TypeScript ;

* la DIV de class form-group est ensuite répété pour chaque FormControl dans le FormArray (retourné par getHobbies() , initialement vide, en notant l'index afin de créer un nom unique pour chaque FormControl ;

* dans cette DIV , vous avec une INPUT qui prendra comme formControlName l'index du FormControl ;

* enfin, vous avez le bouton (de type button pour l'empêcher d'essayer de soumettre le formulaire) qui déclenche onAddHobby() , méthode qui, pour rappel, crée un nouveau FormControl (affichant une nouvelle instance de la DIV de class form-group , et donc créant une nouvelle INPUT)

_____
Interagissez avec un serveur avec HttpClient (FIREBASE)

* [firebase.com](https://firebase.google.com/)

Angular met à disposition un service appelé HttpClient qui

permet de créer et d'exécuter des appels HTTP (fait par AJAX - Asynchronous JavaScript and XML) et de réagir aux informations retournées par le serveur.

Configurer un backend avec le service Firebase de Google.
Ce service permet la création d'un backend complet sans coder, et node comprend énormément de services, dont l'authentification, une base de données NoSQL et un stockage de fichiers.
Dans un chapitre ultérieur, vous apprendrez à utiliser les fonctions mises à disposition par Firebase afin de mieux intégrer le service.

Nous allons simplement utiliser l'API HTTP afin de comprendre l'utilisation de HttpClient .

Préparez le backend

> Firebase > dans Database > Realtime Database.

Revenez à la section Données et notez l'URL de votre base de données, vous allez en avoir besoin pour configurer les appels HTTP :

https://http-client-demo-openclass.firebaseio.com/appareils

et vous allez pouvoir intégrer HttpClient à votre application des appareils électriques.

Envoyez vers le backend

> Dans app.module.ts:
````ts
import { HttpClientModule } from '@angular/common/http';

imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    RouterModule.forRoot(appRoutes)
````

```
],
````

> Vous allez utiliser HttpClient , dans un premier temps, pour la gestion des données de la liste d'appareils. Vous allez donc l'injecter dans AppareilService , en y ayant auparavant ajouté le décorateur @Injectable() (importé depuis @angular/core) :

````ts
import { Subject } from 'rxjs/Subject';
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()

export class AppareilService {

  appareilsSubject = new Subject<any[]>();

  private appareils = [
    {
      id: 1,
      name: 'Machine à laver',
      status: 'éteint'
    }
  ];


  constructor(private httpClient: HttpClient) { }


  saveAppareilsToServer() {     // post()
      this.httpClient

.put('https://http-client-demo-openclass.firebaseio.com/appareils/appareils.json', this.appareils)
        .subscribe(
          () => {
            console.log('Enregistrement terminé !');
          },
          (error) => {
````

```
            console.log('Erreur ! : ' + error);
          }
        );


  getAppareilsFromServer() {    // get()
      this.httpClient

.get<any[]>('https://http-client-demo-openclass.firebaseio.com
/appareils.json')
        .subscribe(
          (response) => {
            this.appareils = response;
            this.emitAppareilSubject();
          },
          (error) => {
            console.log('Erreur ! : ' + error);
          }
        );
// Comme pour  post()  et  put() , la méthode get() retourne
un Observable, mais puisqu'ici, vous allez recevoir des
données, TypeScript a besoin de savoir de quel type elles
seront (l'objet retourné est d'office considéré comme étant un
Object).  Vous devez donc, dans ce cas précis, ajouter
<any[]>

  }
```

Analysez cette méthode :

* la méthode post() , qui permet de lancer un appel POST, prend comme premier argument l'URL visée, et comme deuxième argument le corps de l'appel, c'est-à-dire ce qu'il faut envoyer à l'URL ;

* l'extension .json de l'URL est une spécificité Firebase, pour lui dire que vous lui envoyez des données au format JSON ;

* la méthode post() retourne un Observable – elle ne fait pas d'appel à elle toute seule. C'est en y souscrivant que

l'appel est lancé ;

* dans la méthode subscribe() , vous prévoyez le cas où tout fonctionne et le cas où le serveur vous renverrait une erreur.

> Créez maintenant un bouton dans AppareilViewComponent qui déclenche cette sauvegarde (en passant, si vous ne l'avez pas encore fait, vous pouvez retirer l'activation conditionnelle des boutons "Tout allumer" et "Tout éteindre") :

> Dans appareil-view.component.html:
````
```html
  <button class="btn btn-secondary btn-block mt-5"
  (click)="onSave()">
  Enregistrer</button>

  <button class="btn btn-secondary btn-block mb-5"
  (click)="onFetch()">
  Récupérer</button>
```
````

> Dans appareil-view.component.ts:
````
```ts
  onSave() {
    this.appareilService.saveAppareilsToServer();
  }

  onFetch() {
    this.appareilService.getAppareilsFromServer();
  }
```
````
Il serait également possible de rendre automatique le chargement et l'enregistrement des appareils (par exemple en appelant la méthode getAppareilsFromServer() dans ngOnInit() , et saveAppareilsToServer() après chaque modification)

Créez une application complète avec Angular et Firebase

**Vous allez créer une application simple qui recense les livres que vous avez chez vous, dans votre bibliothèque. Vous

pourrez ajouter une photo de chaque livre. L'utilisateur devra être authentifié pour utiliser l'application.**

Malgré sa popularité, j'ai choisi de ne pas intégrer AngularFire dans ce cours. [GitHub Firebase/AngularFire](https://github.com/firebase/angularfire) Vous emploierez l'API JavaScript mise à disposition directement par Firebase.

Pensez à la structure de l'application

Prenez le temps de réfléchir à la construction de l'application. Quels seront les components dont vous aurez besoin ? Les services ? Les modèles de données ?

Il faudra également ajouter du routing à cette application, permettant l'accès aux différentes parties, avec une guard pour toutes les routes sauf l'authentification, empêchant les utilisateurs non authentifiés d'accéder à la bibliothèque.

Structurez l'application:

Pour cette application, je vous conseille d'utiliser le CLI pour la création des components. L'arborescence sera la suivante :

````bash
ng g c auth/signup
ng g c auth/signin
ng g c book-list
ng g c book-list/single-book
ng g c book-list/book-form
ng g c header
ng g s services/auth
ng g s services/books
ng g s services/auth-guard
````

> Dans app.module.ts importer modules:
````ts
import { FormsModule, ReactiveFormsModule } from
'@angular/forms';
````

```
import { Routes } from '@angular/router';
import { RouterModule } from '@angular/router';
import { HttpClientModule } from '@angular/common/http';

const appRoutes: Routes = [
  { path: 'auth/signup', component: SignupComponent },
  { path: 'auth/signin', component: SigninComponent },
  { path: 'books', component: BookListComponent },
  { path: 'books/new', component: BookFormComponent },
  { path: 'books/view/:id', component: SingleBookComponent }
];

imports: [
    BrowserModule,
    FormsModule,ReactiveFormsModule,
    HttpClientModule,
    RouterModule.forRoot(appRoutes)
],
providers: [
  AuthService, BooksService, AuthGuardService],
````


> Dans models/book.model.ts:
````ts
export class Book {
  photo: string;
  synopsis: string;
  constructor(public title: string, public author: string) {
  }
}
````


> Sans HeaderComponent la navbar:
````html
<nav class="navbar navbar-expand-sm navbar-light  mb-2
bg-light justify-content-center">
  <div class="container-fluid">
    <ul class="nav navbar-nav">
      <li routerLinkActive="active">
        <a routerLink="books">Livres</a>
      </li>
    </ul>
```

```
    <ul class="nav navbar-nav navbar-right">
      <li routerLinkActive="active">
        <a routerLink="auth/signup">Créer un compte</a>
      </li>
      <li routerLinkActive="active">
        <a routerLink="auth/signin">Connexion</a>
      </li>
    </ul>
  </div>
</nav>
```

> Dans AppComponent:
```html
<app-header></app-header>
<div class="container">
  <router-outlet></router-outlet>
</div>
```

### Intégrez Firebase à votre application

D'abord, installez Firebase avec NPM :
```bash
npm install firebase --save
```

Choisissez "Ajouter Firebase à votre application Web" et copiez-collez la configuration dans le constructeur de votre AppComponent (en ajoutant import * as firebase from 'firebase'; en haut du fichier, mettant à disposition la méthode initializeApp() ) :

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

import * as firebase from 'firebase';

export class AppComponent {

  constructor() {
  // Your web app's Firebase configuration
```

```
    const firebaseConfig = {
    apiKey: "AIzaSyCUK66JbpU71b7trxWJNv8c6q3Es7CH_iA",
    authDomain: "openclass-exo-book.firebaseapp.com",
    databaseURL: "https://openclass-exo-book.firebaseio.com",
    projectId: "openclass-exo-book",
    storageBucket: "openclass-exo-book-34.appspot.com",
    messagingSenderId: "1051558273662",
    appId: "1:1051558273662:web:136229eaa3522b449e6e2f"
  };
  // Initialize Firebase
  firebase.initializeApp(firebaseConfig);

}
````

Authentification

L'authentification Firebase emploie un système de token : un jeton d'authentification est stocké dans le navigateur, et est envoyé avec chaque requête nécessitant l'authentification.

Dans Firebase "Configurer un mode de connexion (authentification)

> Dans AuthService , vous allez créer trois méthodes :

* une méthode permettant de créer un nouvel utilisateur ;

* une méthode permettant de connecter un utilisateur existant ;

* une méthode permettant la déconnexion de l'utilisateur.

Puisque les opérations de création, de connexion et de déconnexion sont asynchrones, c'est-à-dire qu'elles n'ont pas un résultat instantané, les méthodes que vous allez créer pour les gérer retourneront des PROMISE, ce qui permettra également de gérer les situations d'erreur.

> Importez Firebase dans AuthService :

````ts
````

```
import { Injectable } from '@angular/core';

import * as firebase from 'firebase';

@Injectable()
export class AuthService {

// Pour créer un nouvel utilisateur, qui prendra comme
argument une adresse mail et un mot de passe, et qui
retournera une Promise qui résoudra si la création réussit, et
sera rejetée avec le message d'erreur si elle ne réussit pas :
  createNewUser(email: string, password: string) {
    return new Promise(
      (resolve, reject) => {
        firebase.auth().createUserWithEmailAndPassword(email,
password).then(
          () => {
            resolve();
          },
          (error) => {
            reject(error);
          }
        );
      }
    );
}// Toutes les méthodes liées à l'authentification Firebase se
trouvent dans  firebase.auth().


// Créez également  signInUser() , méthode très similaire, qui
s'occupera de connecter un utilisateur déjà existant :
signInUser(email: string, password: string) {
    return new Promise(
      (resolve, reject) => {
        firebase.auth().signInWithEmailAndPassword(email,
password).then(
          () => {
            resolve();
          },
          (error) => {
            reject(error);
          }
```

```
        );
      }
    );
}


// Créez une méthode simple  signOutUser()  :
signOutUser() {
    firebase.auth().signOut();
}
````
Ainsi, vous avez les trois fonctions dont vous avez besoin pour intégrer l'authentification dans l'application !

Vous pouvez ainsi créer SignupComponent et SigninComponent , intégrer l'authentification dans HeaderComponent afin de montrer les bons liens, et implémenter AuthGuard pour protéger la route /books et toutes ses sous-routes.

> 1- SIGNUP.Component.TS afin de pouvoir enregistrer un utilisateur :
````ts
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { AuthService } from '../../services/auth.service';
import { Router } from '@angular/router';


export class SignupComponent implements OnInit {

  signupForm: FormGroup;
  errorMessage: string;

  constructor(private formBuilder: FormBuilder,
              private authService: AuthService,
              private router: Router) { }

  ngOnInit() {
    this.initForm();
  }
```

```
  initForm() {
    this.signupForm = this.formBuilder.group({
      email: ['', [Validators.required, Validators.email]],
      password: ['', [Validators.required,
Validators.pattern(/[0-9a-zA-Z]{6,}/)]]
    });
  }

  onSubmit() {
    const email = this.signupForm.get('email').value;
    const password = this.signupForm.get('password').value;

    this.authService.createNewUser(email, password).then(
      () => {
        this.router.navigate(['/books']);
      },
      (error) => {
        this.errorMessage = error;
      }
    );
  }
}
```

Dans ce component :

* vous générez le formulaire selon la méthode réactive

  * les deux champs, `email` et `password`, sont requis — le champ `email` utilise `Validators.email` pour obliger un string sous format d'adresse email ; le champ `password` emploie `Validators.pattern` pour obliger au moins 6 caractères alphanumériques, ce qui correspond au minimum requis par Firebase ;

* vous gérez la soumission du formulaire, envoyant les valeurs rentrées par l'utilisateur à la méthode `createNewUser()`

  * si la création fonctionne, vous redirigez l'utilisateur vers `/books` ;

  * si elle ne fonctionne pas, vous affichez le message d'erreur renvoyé par Firebase.

> Dans SIGNUPComponent.HTML , vous trouverez le template correspondant :

```html
<div class="row">
  <div class="col-sm-8 col-sm-offset-2">
    <h2>Créer un compte</h2>
    <form [formGroup]="signupForm" (ngSubmit)="onSubmit()">
      <div class="form-group">
        <label for="email">Adresse mail</label>
        <input type="text"
               id="email"
               class="form-control"
               formControlName="email">
      </div>
      <div class="form-group">
        <label for="password">Mot de passe</label>
        <input type="password"
               id="password"
               class="form-control"
               formControlName="password">
      </div>
      <button class="btn btn-primary"
              type="submit"
              [disabled]="signupForm.invalid">
              Créer mon compte</button>
    </form>
    <p class="text-danger">{{ errorMessage }}</p>
  </div>
</div>
```

> 2-  SIGNIN.Component.TS & HTML pour la connexion d'un utilisateur déjà existant:

*Vous pouvez créer un template presque identique pour SIGNINComponent  pour la connexion d'un utilisateur déjà existant.  Il vous suffit de renommer  signupForm  en signinForm  et d'appeler la méthode  signInUser()  plutôt que createNewUser() .*

> Modifier HEADERComponent pour afficher de manière contextuelle les liens de connexion, de création d'utilisateur et de déconnexion :

```ts
import { Component, OnInit } from '@angular/core';
import { AuthService } from '../services/auth.service';
import * as firebase from 'firebase';


export class HeaderComponent implements OnInit {

  isAuth: boolean;

  constructor(private authService: AuthService) { }

  ngOnInit() {
    firebase.auth().onAuthStateChanged(
      (user) => {
        if(user) {
          this.isAuth = true;
        } else {
          this.isAuth = false;
        }
      }
    );
  }

  onSignOut() {
    this.authService.signOutUser();
  }

}
// Ici, vous utilisez onAuthStateChanged() , qui permet d'observer l'état de l'authentification de l'utilisateur : à chaque changement d'état, la fonction que vous passez en argument est exécutée. Si l'utilisateur est bien authentifié, onAuthStateChanged() reçoit l'objet de type firebase.User correspondant à l'utilisateur. Vous pouvez ainsi baser la valeur de la variable locale isAuth selon l'état d'authentification de l'utilisateur, et afficher les liens correspondant à cet état.
```

> Dans HEADER.comonent.HTML:
````html
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <ul class="nav navbar-nav">
      <li routerLinkActive="active">
        <a routerLink="books">Livres</a>
      </li>
    </ul>
    <ul class="nav navbar-nav navbar-right">
      <li routerLinkActive="active" *ngIf="!isAuth">
        <a routerLink="auth/signup">Créer un compte</a>
      </li>
      <li routerLinkActive="active" *ngIf="!isAuth">
        <a routerLink="auth/signin">Connexion</a>
      </li>
      <li>
        <a (click)="onSignOut()"
           style="cursor:pointer"
           *ngIf="isAuth">Déconnexion</a>
      </li>
    </ul>
  </div>
</nav>
````

> Il ne vous reste plus qu'à créer AHTHGUARD.Service .Puisque la vérification de l'authentification est asynchrone, votre service retournera une Promise :
````ts
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { Observable } from 'rxjs/Observable';
import * as firebase from 'firebase';

@Injectable()
export class AuthGuardService implements CanActivate {

  constructor(private router: Router) { }

  canActivate(): Observable<boolean> | Promise<boolean> |
````

```
boolean {
    return new Promise(
      (resolve, reject) => {
        firebase.auth().onAuthStateChanged(
          (user) => {
            if(user) {
              resolve(true);
            } else {
              this.router.navigate(['/auth', 'signin']);
              resolve(false);
            }
          }
        );
      }
    );
  }
}
````

> Dans APP.module.ts l'appliquer aux routes concernées PATH
````ts
const appRoutes: Routes = [
  { path: 'auth/signup', component: SignupComponent },
  { path: 'auth/signin', component: SigninComponent },
  { path: 'books', canActivate: [AuthGuardService], component:
BookListComponent },
  { path: 'books/new', canActivate: [AuthGuardService],
component: BookFormComponent },
  { path: 'books/view/:id', canActivate: [AuthGuardService],
component: SingleBookComponent },
  { path: '', redirectTo: 'books', pathMatch: 'full' },
  { path: '**', redirectTo: 'books' }
];

// Ah, mais qu'a-t-on oublié ?  Le routing ne prend en compte
ni le path vide, ni le path wildcard !  Ajoutez ces routes dès
maintenant pour éviter toute erreur :
````

Base de données

Dans ce chapitre, vous allez créer les fonctionnalités de

l'application : la création, la visualisation et la suppression des livres, le tout lié directement à la base de données Firebase.

Pour créer BooksService :

* vous aurez un array local books et un Subject pour l'émettre ;

* vous aurez des méthodes :

  * pour enregistrer la liste des livres sur le serveur,

  * pour récupérer la liste des livres depuis le serveur,

  * pour récupérer un seul livre,

  * pour créer un nouveau livre,

  * pour supprimer un livre existant.

> Dans BOOKS.Service.TS (sans oublier d'importer Book et Subject) :

```ts
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs/Subject';
import { Book } from '../models/book.model';

@Injectable()
export class BooksService {

  books: Book[] = [];
  booksSubject = new Subject<Book[]>();

  constructor() {
    this.getBooks(); //  au démarrage de l'application
  }

  emitBooks() {
    this.booksSubject.next(this.books);
  }
```

```
// méthode mise à disposition par Firebase pour enregistrer la liste sur un node de la base de données – la méthode  set()  :
  saveBooks() {
      firebase.database().ref('/books').set(this.books);
  }
// La méthode  ref()  retourne une référence au node demandé de la base de données, et  set()  fonctionne plus ou moins comme  put()  pour le HTTP : il écrit et remplace les données au node donné.


// méthodes pour récupérer la liste entière des livres et pour récupérer un seul livre, en employant les deux fonctions proposées par Firebase :
getBooks() {
    firebase.database().ref('/books')
      .on('value', (data: DataSnapshot) => {
          this.books = data.val() ? data.val() : [];
          this.emitBooks();
        }
      );
  }
  // Pour  getBooks() , vous utilisez la méthode  on() .  Le premier argument  'value'  demande à Firebase d'exécuter le callback à chaque modification de valeur enregistrée au endpoint choisi : cela veut dire que si vous modifiez quelque chose depuis un appareil, la liste sera automatiquement mise à jour sur tous les appareils connectés.

  // Le deuxième argument est la fonction callback, qui reçoit ici une  DataSnapshot  : un objet correspondant au node demandé, comportant plusieurs membres et méthodes (il faut importer  DataSnapshot  depuis  firebase.database.DataSnapshot ).  La méthode qui vous intéresse ici est  val() , qui retourne la valeur des données, tout simplement.  Votre callback prend également en compte le cas où le serveur ne retourne rien pour éviter les bugs potentiels.

  getSingleBook(id: number) {
    return new Promise(
      (resolve, reject) => {
        firebase.database().ref('/books/' +
```

```
id).once('value').then(
          (data: DataSnapshot) => {
            resolve(data.val());
          }, (error) => {
            reject(error);
          }
        );
      }
    );
  }
```

// La fonction getSingleBook() récupère un livre selon son id, qui est simplement ici son index dans l'array enregistré. Vous utilisez once() , qui ne fait qu'une seule requête de données. Du coup, elle ne prend pas une fonction callback en argument mais retourne une Promise, permettant l'utilisation de .then() pour retourner les données reçues.

```
//  il ne reste plus qu'à créer les méthodes pour la création d'un nouveau livre et la suppression d'un livre existant :
  createNewBook(newBook: Book) {
      this.books.push(newBook);
      this.saveBooks();
      this.emitBooks();
    }

    removeBook(book: Book) {
      const bookIndexToRemove = this.books.findIndex(
        (bookEl) => {
          if(bookEl === book) {
            return true;
          }
        }
      );
      this.books.splice(bookIndexToRemove, 1);
      this.saveBooks();
      this.emitBooks();
    }


// méthode qui permet d'uploader une photo :
    uploadFile(file: File) {
```

```
        return new Promise(
          (resolve, reject) => {
            const almostUniqueFileName =
Date.now().toString();
            const upload = firebase.storage().ref()
              .child('images/' + almostUniqueFileName +
file.name).put(file);

upload.on(firebase.storage.TaskEvent.STATE_CHANGED,
              () => {
                console.log('Chargement…');
              },
              (error) => {
                console.log('Erreur de chargement ! : ' +
error);
                reject();
              },
              () => {
                resolve(upload.snapshot.ref.getDownloadURL());
              }
            );
          }
        );
    }


}
```

> Ensuite, vous allez créer BOOKLISTComponent.TS , qui :

* souscrit au Subject du service et déclenche sa première émission ;

* affiche la liste des livres, où chaque livre peut être cliqué pour en voir la page SingleBookComponent ;

* permet de supprimer chaque livre en utilisant removeBook() ;

* permet de naviguer vers BookFormComponent pour la création d'un nouveau livre.

````ts
import { Component, OnDestroy, OnInit } from '@angular/core';
import { BooksService } from '../services/books.service';
import { Book } from '../models/book.model';
import { Subscription } from 'rxjs/Subscription';
import { Router } from '@angular/router';


export class BookListComponent implements OnInit, OnDestroy {

  books: Book[];
  booksSubscription: Subscription;

  constructor(private booksService: BooksService, private
router: Router) {}

  ngOnInit() {
    this.booksSubscription =
this.booksService.booksSubject.subscribe(
      (books: Book[]) => {
        this.books = books;
      }
    );
    this.booksService.emitBooks();
  }

  onNewBook() {
    this.router.navigate(['/books', 'new']);
  }

  onDeleteBook(book: Book) {
    this.booksService.removeBook(book);
  }

  onViewBook(id: number) {
    this.router.navigate(['/books', 'view', id]);
  }

  ngOnDestroy() {
    this.booksSubscription.unsubscribe();
  }
````

````
}
````

> Dans BOOKLIST.component.HTML:
````html
<div class="row">
  <div class="col-xs-12">

    <h2>Vos livres</h2>

    <div class="list-group">

      <button
        class="list-group-item"
        *ngFor="let book of books; let i = index"
        (click)="onViewBook(i)">

        <h3 class="list-group-item-heading">
          {{ book.title }}
          <button class="btn btn-default pull-right"
(click)="onDeleteBook(book)">
            <span class="glyphicon glyphicon-minus"></span>
          </button>
        </h3>

        <p class="list-group-item-text">{{ book.author }}</p>
      </button>

    </div>

    <button class="btn btn-primary"
    (click)="onNewBook()">
      Nouveau livre</button>

  </div>
</div>
````

> **Il n'y a rien de nouveau ici, donc passez rapidement à SINGLEBOOKComponent.TS :**
````ts
import { Component, OnInit } from '@angular/core';
````

```
import { Book } from '../../models/book.model';
import { ActivatedRoute, Router } from '@angular/router';
import { BooksService } from '../../services/books.service';


export class SingleBookComponent implements OnInit {

  book: Book;

  constructor(private route: ActivatedRoute, private
booksService: BooksService,
              private router: Router) {}

  ngOnInit() {
    this.book = new Book('', '');
    const id = this.route.snapshot.params['id'];
    this.booksService.getSingleBook(+id).then(
      (book: Book) => {
        this.book = book;
      }
    );
  }

  onBack() {
    this.router.navigate(['/books']);
  }
}
```

> Le component récupère le livre demandé par son id grâce à
getSingleBook() , et l'affiche dans le template suivant :
```html
<div class="row">
  <div class="col-xs-12">
    <h1>{{ book.title }}</h1>
    <h3>{{ book.author }}</h3>
    <p>{{ book.synopsis }}</p>
    <button class="btn btn-default"
(click)="onBack()">Retour</button>
  </div>
</div>
```

> **Il ne reste plus qu'à créer BOOKFORM.component , qui comprend un formulaire selon la méthode réactive et qui enregistre les données reçues grâce à createNewBook() :**

````ts
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from
'@angular/forms';
import { Book } from '../../models/book.model';
import { BooksService } from '../../services/books.service';
import { Router } from '@angular/router';


export class BookFormComponent implements OnInit {

  bookForm: FormGroup;

  constructor(private formBuilder: FormBuilder, private
booksService: BooksService,
              private router: Router) { }

  ngOnInit() {
    this.initForm();
  }

  initForm() {
    this.bookForm = this.formBuilder.group({
      title: ['', Validators.required],
      author: ['', Validators.required],
      synopsis: ''
    });
  }

  onSaveBook() {
    const title = this.bookForm.get('title').value;
    const author = this.bookForm.get('author').value;
    const synopsis = this.bookForm.get('synopsis').value;
    const newBook = new Book(title, author);
    newBook.synopsis = synopsis;
    this.booksService.createNewBook(newBook);
    this.router.navigate(['/books']);
  }
````

```
}
````

> Dans BOOKFORM.component.HTML:
````html
<div class="row">
  <div class="col-sm-8 col-sm-offset-2">
    <h2>Enregistrer un nouveau livre</h2>
    <form [formGroup]="bookForm"
    (ngSubmit)="onSaveBook()">

      <div class="form-group">
        <label for="title">Titre</label>
        <input type="text" id="title" class="form-control"
        formControlName="title">
      </div>

      <div class="form-group">
        <label for="author">Auteur</label>
        <input type="text" id="author" class="form-control"
        formControlName="author">
      </div>

      <div class="form-group">
        <label for="synopsis">Synopsis</label>
        <textarea id="synopsis" class="form-control"
        formControlName="synopsis">
        </textarea>
      </div>

      <button class="btn btn-success"
      [disabled]="bookForm.invalid" type="submit">
      Enregistrer</button>

    </form>
  </div>
</div>
````

Storage

Dans ce dernier chapitre, vous allez apprendre à utiliser

l'API Firebase Storage afin de permettre à l'utilisateur d'ajouter une photo du livre, de l'afficher dans SingleBookComponent et de la supprimer si on supprime le livre, afin de ne pas laisser des photos inutilisées sur le serveur.

> Tout d'abord, vous allez ajouter une méthode dans BOOK.service qui permet d'uploader une photo :

````ts
uploadFile(file: File) {
    return new Promise(
      (resolve, reject) => {
        const almostUniqueFileName = Date.now().toString();
        const upload = firebase.storage().ref()
          .child('images/' + almostUniqueFileName +
file.name).put(file);
        upload.on(firebase.storage.TaskEvent.STATE_CHANGED,
          () => {
            console.log('Chargement…');
          },
          (error) => {
            console.log('Erreur de chargement ! : ' + error);
            reject();
          },
          () => {
            resolve(upload.snapshot.ref.getDownloadURL());
          }
        );
      }
    );
}
````

Analysez cette méthode :

* l'action de télécharger un fichier prend du temps, donc vous créez une méthode asynchrone qui retourne une Promise ;

* la méthode prend comme argument un fichier de type File ;

* afin de créer un nom unique pour le fichier (évitant ainsi d'écraser un fichier qui porterait le même nom que celui que

l'utilisateur essaye de charger), vous créez un string à partir de Date.now() , qui donne le nombre de millisecondes passées depuis le 1er janvier 1970 ;

* vous créez ensuite une tâche de chargement upload :

  * firebase.storage().ref() vous retourne une référence à la racine de votre bucket Firebase,

  * la méthode child() retourne une référence au sous-dossier images et à un nouveau fichier dont le nom est l'identifiant unique + le nom original du fichier (permettant de garder le format d'origine également),

* vous utilisez ensuite la méthode on() de la tâche upload pour en suivre l'état, en y passant trois fonctions :

  * la première est déclenchée à chaque fois que des données sont envoyées vers le serveur,

  * la deuxième est déclenchée si le serveur renvoie une erreur,

  * la troisième est déclenchée lorsque le chargement est terminé et permet de retourner l'URL unique du fichier chargé.

Pour des applications à très grande échelle, la méthode Date.now() ne garantit pas à 100% un nom de fichier unique, mais pour une application de cette échelle, cette méthode suffit largement.

> Maintenant que le service est prêt, vous allez ajouter les fonctionnalités nécessaires à BOOKFORM.component .
```ts
// Commencez par ajouter quelques membres supplémentaires au component :
bookForm: FormGroup;
fileIsUploading = false;
fileUrl: string;
fileUploaded = false;

// Ensuite, créez la méthode qui déclenchera  uploadFile()  et
```

qui en récupérera l'URL retourné :

```
onUploadFile(file: File) {
    this.fileIsUploading = true;
    this.booksService.uploadFile(file).then(
      (url: string) => {
        this.fileUrl = url;
        this.fileIsUploading = false;
        this.fileUploaded = true;
      }
    );
}
```

// Vous utiliserez fileIsUploading pour désactiver le bouton submit du template pendant le chargement du fichier afin d'éviter toute erreur – une fois l'upload terminé, le component enregistre l'URL retournée dans fileUrl et modifie l'état du component pour dire que le chargement est terminé.

// Il faut modifier légèrement onSaveBook() pour prendre en compte l'URL de la photo si elle existe :

```
onSaveBook() {
    const title = this.bookForm.get('title').value;
    const author = this.bookForm.get('author').value;
    const synopsis = this.bookForm.get('synopsis').value;
    const newBook = new Book(title, author);
    newBook.synopsis = synopsis;
    if(this.fileUrl && this.fileUrl !== '') {
      newBook.photo = this.fileUrl;
    }
    this.booksService.createNewBook(newBook);
    this.router.navigate(['/books']);
}
```

// Vous allez créer une méthode qui permettra de lier le <input type="file"> (que vous créerez par la suite) à la méthode onUploadFile() :

```
detectFiles(event) {
    this.onUploadFile(event.target.files[0]);
}
```

// Il faut également prendre en compte que si un livre est supprimé, il faut également en supprimer la photo. La

nouvelle méthode removeBook() est la suivante :

```
removeBook(book: Book) {
    if(book.photo) {
      const storageRef =
firebase.storage().refFromURL(book.photo);
      storageRef.delete().then(
        () => {
          console.log('Photo removed!');
        },
        (error) => {
          console.log('Could not remove photo! : ' + error);
        }
      );
    }
    const bookIndexToRemove = this.books.findIndex(
      (bookEl) => {
        if(bookEl === book) {
          return true;
        }
      }
    );
    this.books.splice(bookIndexToRemove, 1);
    this.saveBooks();
    this.emitBooks();
}
````

> Dans BOOKFORM.component.HTML L'événement est envoyé à cette méthode depuis cette nouvelle section du template :

````
```html

    <h4>Ajouter une photo</h4>
    <input type="file" (change)="detectFiles($event)"
           class="form-control" accept="image/*">
    <p class="text-success" *ngIf="fileUploaded">Fichier
chargé !</p>

<button class="btn btn-success" [disabled]="bookForm.invalid
|| fileIsUploading"
      type="submit">Enregistrer
</button>
<!-- Dès que l'utilisateur choisit un fichier, l'événement est
```
```

déclenché et le fichier est uploadé.  Le texte "Fichier chargé !" est affiché lorsque  fileUploaded  est  true , et le bouton est désactivé quand le formulaire n'est pas valable ou quand  fileIsUploading  est  true . -->
````

> Il ne reste plus qu'à afficher l'image, si elle existe, dans SINGLEBOOK.Component :
````html
<div class="row">
  <div class="col-xs-12">
    <img style="max-width:400px;" *ngIf="book.photo"
[src]="book.photo">
    <h1>{{ book.title }}</h1>
    <h3>{{ book.author }}</h3>
    <p>{{ book.synopsis }}</p>
    <button class="btn btn-default"
(click)="onBack()">Retour</button>
  </div>
</div>
````

Puisqu'il faut une référence pour supprimer un fichier avec la méthode delete() , vous passez l'URL du fichier à refFromUrl() pour en récupérer la référence.

Connection Firebase

Realtime Database>rules>read, write >true

````js
{
  /* Visit https://firebase.google.com/docs/database/security
to learn more about security rules. */
  "rules": {
    ".read": true,
    ".write": true
  }
}
````
````

Application prête à être distribuée:

````bash
ng build --prod
````

Voici les nouvelles compétences Angular pour créer une application dynamique, comportant plusieurs components qui :

* affichent des données dans le template avec le data binding ;

* sont construits de manière dynamique avec les directives ;

* communiquent ensemble grâce aux services ;

* sont accessibles par un routing personnalisé ;

* emploient des Observables pour gérer des flux de données ;

* utilisent des formulaires pour exploiter des données fournies par l'utilisateur ;

* fonctionnent avec un backend Firebase pour la gestion de l'authentification, des données et des fichiers.

_____
Code scaffolding (échaffaudage)
Run `ng generate component component-name` to generate a new component. You can also use `ng generate directive|pipe|service|class|guard|interface|enum|module`.

_____
Build
Run `ng build` to build the project. The build artifacts will be stored in the `dist/` directory. Use the `--prod` flag for a production build.

_____
Running unit tests
Run `ng test` to execute the unit tests via [Karma](https://karma-runner.github.io).

_____
Running end-to-end tests
Run `ng e2e` to execute the end-to-end tests via
[Protractor](http://www.protractortest.org/).

_____
Further help
To get more help on the Angular CLI use `ng help` or go check
out the [Angular CLI
README](https://github.com/angular/angular-cli/blob/master/REA
DME.md).