1. **TypeScript**

* C’est un super Set du JavaScript
* C’est du JavaScript compile
* Permet de **limiter les** **erreurs : Uncaught TypeError**
* Meilleur autocomplétions et documentation
* Le TypeScript peut être convertis en **ES3 ou ESNext**
* **Installation de TypeScripte**
* Initialiser le projet : **npm init -y**
* Installer la dépendance typeScript : **npm install typescript ---save-dev**
* Compilation de code typeScript en JavaScript :
* npx tsc nom\_fichier\_typeScript
* npx tsc nomFichier --outDir dist ( **--outDir dist :** signifie qu’ on crée un dossier  **dist** pour stocker le fichier javascript)
* On peut pour faciliter les choses crée un fichier : **tsconfig.json** qui va contenir les configuration

{

‘’compilerOptions’’ :{

‘’outDir’’ : ‘’dist’’ // dossier de destination

‘’target’’ : ‘’ES2021’’ // cible de compilation

‘’noEmitOnError’’ :true // cas d’erreur ne convertis pas le code

‘’strict’’ : true //typage et autres choses actives

},

‘’files’’ :[ // liste des fichiers à charger

‘’path fichier ts’’

]

}

* "strictPropertyInitialization": false, : permet de créer les variables sans devoir les initialisées directement. **Ceci est ajouter dans le fichier tsconfig.json**
* Pour lancer la compilation avec un fichier **tsconfig.json** :
  + npx tsc
  + npx tsc --watch lance en mode observeur, c’est à dire toutes les modifications seront actualisées automatiquement.
* **Syntaxe de base**
* nomVariable : type = valeur
  + **Exple :** const a : string = ‘hello world’ ;
* Les Objets :
* const **user** :{**firstN** : string, **lastN?**: string}={‘Paul‘, ‘L‘}
* **?:** signifie que cette variable sera optionnelle.
* **as** permet de forcer le type d’une variables
* **<type>** permet aussi de forcer le type d’une variable
* **type1|type2 :** ce type **union**  permet de soit être du **type1** ou du **type2**
* **variable?.** ⬄ **if(variable !=0) :**  permet de savoir si une variable n’est pas nul ou existe
* **Narrowing :**

système qui permet de réduire les types pour les variables

* On peut avoir du **Narrowing**  grâce à : **typeof nomVariable**
* On peut aussi avoir avec : **variable instanceof**

**Exple :**  if(age instanceof Date){…..}

* variable! : permet de dire que **variable ne pourra jamais être nul**.
* **Alias et Generic**
* **Alias de type :**

C’est un raccourci vers un autre type

//Création du type

type User = {firstName:string, lastName: string};

//Utilisation

const user: User ={firstName: 'John', lastName: 'Doe'};

On peut également faire des **Alias** pour des types **Basic (**number, string, etc…**)**

//Création du type

type DateString = string;

type Id = string|number; // pour les union

//Utilisation

const date:DateString = 'string';

const id: Id = 2;

* **Generic :**

Elle permet de définir des fonctions ou des variables qui peuvent prendre n’importe quel type. On le note : **function<Type>(…)**

Nous voyons ici que nous avons défini une fonction avec un type Generic Type, mais pendant l’utilisation nous lui fournissons un type bien précis. Pour une même fonction on peut générer plusieurs types.

//Création de la fonction

function identity<Type>(arg: Type): Type{

    return arg;

}

//Utilisation de la fonction

const val = identity<number>(3);

* Pour les Tableau on a :
  + Le type Generic **Array :**

const tab: Array<number | string> = ["Paul", "Jean", 3];

* **Les classes :**
* Les **spread**  permettent de copier tous les éléments d’un tableau dans un autre tableau : **[…tab]**

1. **Bootstrap**
2. **Breakpoint**

* **aucun:** Petit Smartphone ( **< 576 px** )
* **sm:** Smartphone ( **≥ 576 px** )/ transition être mobile et Tablette
* **md:** Tablette ( **≥ 768 px** )
* **lg:** grande tablette et Pc 13 ‘’ ( **≥ 992 px** )
* **xl :** écran 15’’ ( **≥ 1200 px** )
* **xxl :** écran supérieur à 15’’ ( **≥ 1400 px** )

**Exple:**

**Expli:**

* la classe **.col-6** signifie que pour tous les écrans sans restriction de taille minimum ou maximum, attribuez **06 colonnes** à cette <div>. **Pour petit écran/smartphone** ;
* la classe **.col-md-4** signifie que pour tous les écrans avec une largeur minimum supérieure ou égale à 768 px, attribuez **04 colonnes** à cette <div>. **Pour moyen écran/Tablette** ;
* la classe **.col-lg-3** signifie que pour tous les écrans avec une largeur minimum supérieure ou égale à 992 px, attribuez **03** **colonnes** à cette <div>. **Pour grand écran/Pc**.

**NB:** L’utilisation d’un modificateur (**Breakpoint**) sur une <div> doit aussi être utiliser sur l’autre <div> de telle manière a obtenir 12 colonnes.

Donc : **.col-md-4 +.col-md-8 = .col-md-12**

1. **Gouttière(gutters)**

C’est l’espacement entre les différentes colonnes de la grille.

* **gy-\* :**  espacement verticale
* **gx-**\* : espacement horizontal
* **g-**{1-5} : plage des valeurs des espacements
* **g-**4 : espacement par défaut
* **offset-**{0-11} : permet de pousser de **0 à 11 colonnes**  le contenu de la colonne sur laquelle on ajoute cette classe.

1. **Angular**

Les composantes Angular sont composées de :

* Template (HTML)
* Contrôleur (JavaScript/TypeScritpt)
* Métadonnées

1. **Installation**

* **NPM** (Node Package Manager): est une bibliothèque en ligne regroupant plusieurs code utilitaire JavaScript, TypeScripte et CSS.
* **npm install @angular/cli** 🡪 permet d’installer le CLI de Angular.
* **npm install -g @angular/cli** (installe globalement Angular CLI).
* **npm -v** 🡪 Donne la version
* **ng v 🡪** Donne la version de Angular.
* Création d’un projet avec CLI : **NB :** se place dans le dossier voulu :
* **ng** new nomProjet --defaults ( **ng :** le nom de Angular CLI. **nomProjet :** est le nom du projet. **--defaults**: créé un projet simple)
* **npm run start :** permet de lancer Angular
* Structuration d’un fichier Angular :
* Dossier **« node Module » :**  contient l’ensemble des paquets utilises par Angular.
* Dossier **« Src »**  c’est le dossier du programme en cours et contient :
* **Dossier** « **app** » : contient les fichiers pour construire les components qui seront afficher dans le navigateur.
* Dossier **« assert » :** Permet d’ajouter les **images, police d’écritures etc….**
* Dossier **« environments » :**  Permet d’utiliser différent destination **Url**  pour le serveur.
* **« favicon.ico »** : Permet de changer l icone de l’application web.
* **« index.html » :** fichier principale qui charge le contenu de notre application sur le web.
* **« main.ts »** : Permet de démarrer notre application avec la fonction « **bootstrap module »** qui va charger le module principal et qui lui chargera d’autres modules.
* **« polyfills.ts » :** Permet améliorer la capacite du navigateur.
* **« style.css »** : Pour la declaration des styles css.
* **« test.ts »**: Pour faire le lancement des test.
* **«. editorconfig » :** Permet de définir les règles de formatage du code. Par exemple **: alignement, les espacements, etc…**
* **« .browserslistrc » :** déclare de nombre de version du navigateur pouvant supporter l’application compilée.
* **« karma.conf.js » :**  les configurations pour nos fichiers test unitaires.
* **« package.json » :**  Sauvegarde la liste des dépendances de notre application.

**Exple : "start": "ng serve -o"**­ 🡪 lors du démarrage de l’application, il ouvre le navigateur par defaults avec.

Les composantes Angular :

Controlleur TypeScript

Metadonnees + Templates

Import/Telechargement des

Metadonnees

Templates = HTML +CSS

1. **Amorçage de l’App (Bootstrapping)**

Apres création de la composante, il faut maintenant :

* **Faire communique avec le navigateur (index.html)**
* **Enregistrer au processus de démarrage de Angular (app.module.ts)**
* Le code html présent dans le composant template va être appelé grâce a 🡪 **selector : ‘app-root’**

**selector : ‘app-root’**

**Fig. Fichier index.html**

L’élément html **‘app-root’** créé n’existe pas encore dans les modules angular. Pour donc le faire, Angular passe par le module **AppModule(fichier app.module.ts)**  et suis les étapes suivant :

* Organise les composantes
* Création d’un environnement pour gérer les balises html.
* Faire fonctionne l’application dans le navigateur

1. **Template (html) :** interpolation et intégration des librairies

* Template en ligne (**inline**) : sur plusieurs lignes on utilise **backstick**: **´ ´**
* Template avec un chemin d acces (**templateUrl**) :
* **templateUrl : ‘./app.component.html’**
* **Ajouter la bibliothèque de Bootstrap**
* npm install bootstrap --save : ceci va télécharger les paquets bootstrap et ajoute dans le fichier package.json.
* Ensuite allez dans **src/styles.css** pour ajouter le code suivant :

**@import ‘~bootstrap/dist/css/bootstrap.min.css’ ;**

* Ajouter dans le **app.component** le **templateUrl** vers le html**:**
* templateUrl : ‘./app.component.html’
* crée ensuite un nouveau dossier à l’intérieur de **/app** (hotel-list)
* créé un fichier typeScript et html (**hotel-list.component.ts et hotel-list.component.html**) à l’intérieur de **/hotel-list**
* Remplir le fichier typeScript comme suit :
* Aller dans **app.component.html** et appelé le sélecteur précédemment créé :

<app-hotel-list></app-hotel-list>

* Allez dans **app.module.ts**  pour enregistrer notre composante au processus d’amorçage.
* Allez dans **déclarations** et ajouter **HotelListComponent.**
* Ajouter son importation :

import {HotelListComponent} from './hotel-list/hotel-list.component';

1. **Directive Structurelles :** **ngFor et ngIf**

* **Interpolation :**  est le processus qui consiste a utilisé **un attribue ou méthode d’une classe typeScript** dans un Template html. Cela ce fait a l’aide de : **{{….}}**
* **Directive ngIf :**

<h2 \*ngIf="hotels && hotels.length > 0">{{title}}</h2>

**Exple:**

**Expli: hotels** est une variable typeScript, donc ici on affiche le titre des hotels si et seulement si **hotels existe**  et s il  **contient quelque chose.**

* **Directive ngFor :**

<div class="col mb-4" \*ngFor="*let* hotel of hotels">

**Exple:**

**Expli:** Ici cette directive nous permet de dupliquer de bout de code. La variable **hotel** va être appelé dans du html et en se servant du **binding/interpolation** on va pouvoir accéder a aux attributs de l’objet  **hotels.**

**NB: for…of** vs.  **for…in**

1. **Databinding : Property et Event Binding**(**One-Way-Binding**)

**One-Way-Binding**

Ici nous voyons que **one-way-binding**  est unidirectionnel. L envois d’une propriété par l’utilisateur au navigateur et ensuite le navigateur répond via un évènement sous forme par exemple de clic, etc… Par l’utilisateur

* **Exple de Property Binding:**  sur l’image

<img src="{{hotel.imageUrl}}" class="card-img-top" alt="...">

* **Exple de Property Binding avec cible et source**
* **Exple de Event Binding avec cible et source :**

<button class="btn btn-primary" (click)="toggleIsNewBadge()">

* **Installation des Icones :** Materials Icon de Google
* **npm install material-icons --save**
* **Faire son importation dans le fichier style.css :**

@import '~material-icons/iconfont/material-icons.css';

1. **Two-way Data Binding : [ngModel]**

* Cette propriété est le plus souvent utilisée dans les formulaires
* Avant son utilisation il faut d’abord aller **importe**  les **FormsModule**  dans le ficher **app.module.ts**
* On peut par la suite utiliser cette propriété dans notre formulaire
* On doit maintenant générer **l’évènement : (ngModelChange).** Mais Angular rejette en interne un évènement : **$event**

**[ngModel] = ‘variable’ (ngModelChange)=’variable =$event’**

<input type="text" class="form-control" [ngModel]="hotelFilter" (ngModelChange)="hotelFilter = $event">

* On peut se servir de **« la banane dans le boxe ».**  Dans ce cas on ne se sert plus de **$event**, mais plutôt ceci :

**[(ngModel)] = ‘variable’.**

<input type="text" class="form-control" [(ngModel)]="hotelFilter">

**NB** : **Banane dans le box** est le plus utilise

1. **Pipe : ( | )**

* Il transforme les propriétés avant de les afficher
* Nous avons comme **pipe :**
* **date:** transforme en format date
* **currency :** transforme en **unité de monnaie**
* **uppercase**
* **lowercase**
* **percent**
* **Decimal**
* **json:**  transforme en  **objet json**
* **slice, async etc…**
* On peut aussi créer nos propres Pipe personnalisés.
* **Exple de Pipe :**
* **Sur le Pipe currency**  nous avons :
* **Code :**  ‘EUR’
* **Symbole :** défini sous quelle forme la valeur doit être
* **‘1.2 – 2’ :** 
  + - **1 :**  au minimum 1 chiffre avant la virgule
    - **.2 :**  au minimum 2 chiffres après la virgule
    - **- 2 :**  au maximum 2 chiffres après la virgule
* **‘fr’ :**  affichage du prix sous écriture française.
* **Ajouter une langue pour les fonctions native :**
* Aller dans **app.module.ts,**  de l’importer le fichier **registerLocaleData :**

**import { registerLocaleData } from '@angular/common';**

* Importer la langue souhaite (**français**) :

**import localeFr from '@angular/common/locales/fr';**

* Ensuite on enregistre la langue française dans **registerLocaleData() :**

**registerLocaleData(localeFr, 'fr');**

1. **Les interfaces**

* Définissent un ensemble de propriété et méthodes.
* Les classes utilisant ses interfaces doivent les  **implémenter.**
* Elle est utilisée comme **type**  pour les propriétés et méthodes.
* Déclaration d’une interface se fait comme suit  :
* Pour importer cette interface on fait comme suit :
* Pour utiliser cette interface :

1. **Cycle de vie(lifecycle Hook) : ngOnit, ngOnChange ….**

* Création de la page : **ngOnInit(), ngOnChange()**
* Rendu de la page : **ngAfterContentinit(), ngAfterContentChecked()**
* Rendu des sous composants ou composant enfant : **ngAfterViewInit(), ngAfterViewChecked()**
* Gestion des changements pendant l’utilisation de ses composants : **ngOnChanges(), ngDoCheck()**
* Destruction du composant après fermeture de la page : **ngOnDestroy()**
* **Ajout ou l’utilisation d’un niveau de vie dans le composant :**
* **Implémenter l’interface concernée**  en utilisant le mot clé **implements** suivi du niveau de vie
* **Importer l’interface concernée**
* **Implémenter ensuite la méthode de l’interface.**

**Fig. Ajout ou l’utilisation d’un niveau de vie dans le composant**

1. **Encapsulation des feuilles de styles CSS dans les composants**

* En Angular on peut se servir de la propriété **styles** du décorateur **@component({…})**

**Exple : styles : [‘.filter{font-size :30px}’]**

* On peut également utiliser un fichier externe, on va se servir de la propriété **styleUrls : .**

1. **Get, Set avec typeScript**

Pour une **variable privée,** la notation se fait avec : **‘\_’**.

Par exemple : **\_variablePrive**

* **Get** permet la lecture unique de la valeur de la variable
* **Set**  permet de modifier la valeur de la variable
* L’utilisation du Set se fait par une effectuation. Par exemple :

**hotelFilter = „nouveau hotel“** -> pour le **set**

**console.log(hotelFilter)** -> pour le **get**

1. **Création d’un Pipe Personnalisé**

* **Création**
* Créer une classe typeScript
* Ajouter le décorateur **@Pipe({….})**
* Implémenter l’interface d Angular qui permet de créer les Pipes personnalise : **PipeTransform**
* Importer ce PipeTransform et le Pipe du module @angular/core
* Implémenter les méthodes de l’interface qui va avec : **transform(valeur) : string{….}**

**Fig. Création d’un Pipe Personnalisé**

* **Intégration dans le module (app.module.ts)**
* Enregistrer le Pipe dans **Array** des déclarations : **ReplaceComma**
* **Pratique :**
* Créer un nouveau dossier (**shared**) dans le **dossier app**
* Dans le dossier shared créer un autre dossier (**pipes**)
* Dans ce dossier, créer un fichier typeScript (**replace-comma.pipe.ts)**
* Ensuite suivre les étapes de **création et d’intégration du pipe**

**Rq : !!variable**-> permet de savoir si la variable n’est **pas undefine ou vide**

**Exple : !!nom ->**  teste si nom n’est pas undefine ou vide

1. **Construction des composants enfants**

* Chaque composante **parente** pour communiquer avec une composante **enfant**, le fait à l’aide de : **@Input().** Les classe ici **implémenterons toujours l’interface OnChanges**  et la valeur venant du parent ne pourra être manipuler que dans la méthode **ngOnChanges()**
* L’**enfant**  quant à lui pour communiquer avec son **parent** se sert de la fonction **@Output()** qui est un **Evènement**  qui sera envoyé au parent.

* **Proj :Continuions avec notre projet**
* Créer un dossier **components** dans le dossier  **shared**
* Dans le dossier components créer un dossier **star-rating**
* Dans star-rating créer un fichier html et css **star-rating.component.html et star-rating.component.css**
* Créer enfin un contrôleur typeScript **star-rating.component.ts**
* Après avoir déclaré tout dans nos différents fichiers, nous devons maintenant déclarer la composante dans le fichier **app.module.ts**
* **Communication du parent vers enfant : @Input()**
* Nous voulons envoyer une **propriété/variable** du parent vers l’enfant, alors nous allons nous servir du  **Property binding**
* On voudrait modifier la une **propriété css** de l’enfant, alors on se fait de la formule de JavaScript : **style.propriete = valeur.** Avec **Property binding** ça donne **[style.propriete] = ‘’valeur’’**
* Créons une propriété/variable dans la classe du parent et qui sera transmis à l’enfant.
* Il faut maintenant  **décorer** la propriété/variable enfant qui va recevoir celle de son parent avec le décorateur : **@Input()** au-dessus de la propriété
* Enfin on peut maintenant attribut la valeur de la propriété du parent à celui l’enfant au niveau de son **sélecteur html** **situe dans celui du parent** à l’aide du Property binding : **[propriete\_enfant\_decore] = ‘’propriete\_parent ‘’**
* **Communication de l’enfant vers le parent : @Output()**
* La propriété qui sera décorée par  **@Output()**  sera de type  **EventEmitter<..>.** C’est une **classe** qui dispose d’une
* méthode **emit(..).** Cette méthode recevra la propriété de l’enfant : **objetEventEmitter**.**emit(propriete\_enfant)**

**NB :**

* en se servant des **backtick : ` `** , on peut appeler la propriété avec **$** comme suit : **`$propriete\_enfant** **` .**
* Pour une **expression** on se sert des **{} :** `**{$propriete\_enfant +2** **}**`
* **Cote parent**, sur le son html niveau du **selector enfant**, on va créer un **Event Binding**  avec propriété décorée par **@Output()** et une méthode qui prendra en paramètre : **$event.** Cette dernière retournera le message de l’enfant. **$event** contiendra **argument**  contenir dans la methode **emit()** de l’enfant et certains autres paramètres du DOM.

**(propriete\_enfant\_decore) = ‘’ methode\_parent ($event)’’**

* **Methode\_parent(…)**  sera bien sûr définie dans le fichier typeScript du parent.

Pour le projet nous allons procéder ainsi :

* **Cote Enfant :**
* 1- identification de l’élément déclencheur de l’évènement vers le parent
* 2- créer la méthode qui sera exécutée lorsque l’évènement sera actif
* **Cote Parent :**
* 3- le parent va écouter l’événement généré par l’enfant grâce à un **Event Binding**
* 4- déclaré une méthode qui recevra l’événement

1. **Services**

Un **service Angular**  est une classe a responsabilité particulière.

Avec un service on peut :

* Interagir avec des sources de données extérieurs (**base de données)**
* Partager les sources de données avec plusieurs composant
* Ses composants peuvent agir de façon indépendante tout comme le service en lui-même.
* **Caractéristiques :**
* C’est une classe **typeScript**
* Les composants peuvent s’en servir, en instanciant simplement l’objet : **const serviceA = new ServiceA() ;**
  + ***Inconvénient :* Le service ne sera présent que dans le composant sélectionné**
* La meilleur methode est l’utilisation d’un  **injecteur**
* **L’injecteur**  enregistre le service et partage avec tout composant qui voudrait bien utiliser le service.
* Le composant doit créer un constructeur, ensuite ajoute la dépendance nécessaire à partir de l’injecteur : **dependency injection**
* Angular peut enregistrer plusieurs services dans un **injecteur**.

Fig. Processus d injection

Fig. Different Injecteurs

* **Root Injector vs Component Injector**

**Component Injector**

* Disponible seulement aux composant et composant enfant
* Isole le service pour éviter la mauvaise utilisation

**Root Injector**

* Disponible partout et pour tous les cas
* **Création d’une service :**
* Créer une classe typeScript
* Ajouter un décorateur au-dessus de la classe **@Injectable()**
* Importer le décorateur
* Enregistrer ensuite le service dans **providers** situe dans le fichier **app.module.ts :**

providers: [HotelListService]

* **!!!** Cette façons de faire n’est plus utiliser, on utilise plutôt celle qui suit
* **On utilise plutôt** :  **proverdedIn : ‘root’** dans le décorateur **Injectable() :**

@Injectable({

    providedIn: 'root'

})

* + Cette nouvelle methode améliore le processus du  **thread checking :**  processus pendant lequel Angular évacue le code non utilisé pour pouvoir donne un code plus fin pour le site final.

**NB :** On peut aussi utiliser le **providers**  directement dans notre composant :

* **Processus d’injection dans le composant**
* Créer une variable locale dans le composant
* Créer un constructeur et à l’intérieur du constructeur, on passe le paramètre de la  **dépendance** concernée (**Service**)
* Attribue à la variable la valeur du paramètre du constructeur.

**NB :**  Tous ces étape peuvent résumer comme suit :

**(param** peut être: public/protected**)**

**constructor(private param : ServiceClass){…}**

*constructor*(private *hotelListService*: HotelListService){}

**-Exple :**

* **Proj :Continuions avec notre projet**
* Créer un fichier **hotel-list.service.ts**  dans le dossier hotel-list
* Dans ce fichier **crée**  service et faire son injection dans le composant

1. **RxJS et les Observables**

Fig. Communication **http** vers le serveur

* **Rôles du paquet Http d’ Angular :**
* Le paquet **http**  en Angular permet donner les méthodes qui permettent d’envoyer des données a un service web ou d’appeler ces données.
* Permet de recommencer une requête si celle a échoué ou d’intercepter les requêtes non autorisée par notre application.

Les paquets **Http, Form, Router**  d’Angular sont basés sur la bibliothèque JavaScript **RxJS.**

Cette bibliothèque permet  **d’implémenter la programmation réactive avec les observables ou observable dans les applications JavaScript.**

* **Les Observables :**
* **Les Observables sont des opérations Asynchrone**
* Permettent de manager les éléments qui arrivent de façon asynchrone sur le temps
* Ces éléments sont traités en tant que **Array**  ou **collection** et arriverons progressivement en fonction du temps sur une durée donnée.
* **Exemple :**

Dans cette exemple on a une  **séquence d observable :**1, 2, 3, 4 et 5

Les utilisateurs **jean et Helene** voudraient recevoir ces données, tout d’abord ils devront  **s’enregistrer**  en appelant la methode **subscriber().** Quand ils ne voudront plus recevoir de données alors ils devront appelés la methode **unsubscribe().**

* **Operateurs sur les Observables :**
* Permettent de transformer un observable en un nouveau observable
* Permettent de manipuler l’observable existant en fonction de l’opérateur choisi
* Permettent de transformer chaque valeur dès son arrive dans la séquence avant que celle-ci ne soit émise aux différents composants qui ont souscrit à la notification

**Exple : map(), filter(), concat(), find(), take() , etc…**

Ici la methode map() est appelée et prend chaque valeur de la séquence pour la multipliée et ainsi retourne une nouvelle séquence

* [**https://rxmarbles.com/#map**](https://rxmarbles.com/#map) ce site aide pour comprendre ces méthodes.
* [**https://rxviz.com/**](https://rxviz.com/) ce site permet de visualiser les séquences d observables.
* La methode **pipe()** permet de composer ces operateur ainsi les écris les uns a la suites des autres.
* Les variable observables sont note avec **$ :**

**const source$ : Observable<number> ;**

* **Exemple** de code sur **rxviz.com :**

La methode **take()**  prend le nombre d’élément dans notre liste et retourne une nouvelle liste [**0, 1, 2, 3]**, ensuite il transmet cela a la methode **map()** qui a son tour affiche les personne en fonction des valeurs reçu de la nouvelle liste. **persone[0], persone[1], …**

* **Promise vs Observable**

**Promise**

* **.then()… .catch()**
* Une seule valeur en retour
* Ni modifiable et ni annulable

**Observable**

* **.subscribe()**
* Plusieurs valeurs avec le temps.
* Modifiable, annulable.
* Operateur connus sur Array utilisable aussi sur observable et d’autres opérateurs supplémentaires

1. **HttpClient et Error Handling avec RxJS**

* Pour recevoir les données d’un Serveurs dans notre application web nous avons besoin du **service HttpClient**  de Angular
* La réponse va être envoyé à ce service et ce dernier va le transforme en **Observable** et ensuite le retourne a notre navigateur.

**Comment faire pour envoyer une requête à partir de notre service Angular ?**

* **Envois de Requête à partir du service angular :**
* Spécifier **URL** à partir duquel on doit lire les données
* Injecter le service Angular qui est responsable de l’envoie des requêtes : ici c’est **HttpClient**
* Importer l observable de : **rxjs**
* Ensuite on peut maintenant envoyer nos requêtes.
* **Etape pour L’utilisation du Service HttpClient**
* Ajouter dans **imports :** **[HttpClientModule] ,** c’est un module pour utiliser le service **HttpClient**  d’Angular. Il se trouve dans **app.module.ts**
* **Proj :Continuions avec notre projet**
  + Créer un nouveau dossier **api**  dans le dossier **src**.
  + Ajouter ce dossier dans le fichier **angular.json** pour qu’ il soit pris en compte par Angular. Dans ce fichier au niveau de  **assets - ‘’assets’’ :[…., ‘’ src/api ’’].**
  + A l’intérieur du dossier api, on crée un **fichier json :** hotels.json
  + Ce fichier json contiendra la liste des hotels
  + On doit maintenant **injecter** notre service HttpClient dans le service HotelListService
  + Après tout cela il faut relancer le serveur car nous avons effectué des modifications au **angular.json**
* **Intercepter les Erreurs pendant le processus de requêtage :**
* Methode **catchError(methode) :** C’est un **opérateur** qui prend en paramètre une **methode** et cette methode nous aide à capture toutes les **erreurs**  qui peuvent arriver.
* Cette **methode** devra être implémentée
* [**https://angular.io/guide/http#getting-error-details**](https://angular.io/guide/http#getting-error-details) c’est un lien vers l’implémentation de la methode **handleError(…)**
* **Souscription a l observable :**

Pourque la requête **http** soit envoyée , il faut que:

* On appelle **subscribe(…)**  sur l Observable dans le composant.
* **Observable.subscribe(Observer),**  la methode subscribe(…) prend un paramètre optionnel un **observeur.**
* **Observeur** est un objet qui compose plusieurs fonction :
  + **nextFn :**  cette fonction permet de recevoir l’objet qui a été émise à travers cette observable
  + **errorFn :** retourne une **erreur** chaque fois qu’ il y aura rejet sur Observable.
  + **completeFn :**  c’est la fonction qui va être retourne chaque fois qu’ il n’aurait plus d’éléments dans la liste d observable ou que le dernier élément arrivera.
* Avant d’utiliser **unsubscribe(),**  on doit sauvegarder la souscription dans une variable
* **Différence entre this dans la fonction fléché et this fonction simple**
* Dans le **fonction fléché,**  **this**  est **globale** donc fait référence **l’objet d’une instance de la classe.**
* Dans une  **fonction simple, this**  est **locale** donc fait référence à tout objet local a la fonction.

1. **Navigation avec routerLink et route-outlet**

* Tout part d’une page représentée par le **component principale : app-root**
* A l’intérieur de ce composant on va charger toutes les pages de notre application
* Le chargement de de ses pages sont bien sûre effectue par la  **directive de routage : <router-outlet></router-outlet>**
* **router-outlet** permet d’effectuer les changements qui ont été faits par l’utilisateur et de charger à travers le **<ng-component>** le contenu du composant qui a été sélectionné en fonction de la requête
* **Configuration des routes**
* Configurer l’itinéraire pour atteindre chaque route
* Configurer le route pour chaque composant
* Définir une action pour que cette route s affiche
* Attacher un itinéraire a chaque action
* Activer la route en fonction de l’action
* **Enregistrement du service pour les route**
* Il faut enregistrer le service D’Angular qui s’occupe du route :**RouterModule.forRoot([ ])**
* Enregistrer ce service **l’injector** dans le **app.module.ts** au niveau des **imports :[…., RouterModule.forRoot([ ]) ]**
* **RouterModule.forRoot([ liste des route])** il nous permet d’inscrit la liste de nos **chemin url ou route**
* On note les  **routes**  la comme suit :

**{path :’name\_path’, component :name\_component}**

* **{path :’ ’, redirectTo :’homePage’, pathMatch :’full’}** ceci me permet de me rediriger vers  **la page d’accueil.**
* **Proj :Continuions avec notre projet**
* créer un nouveau composant **home**  avec la commande : **ng g c home**
* créer un nouveau composant **hotel-detail** dans le dossier hotel-list : **ng g c hotel-list/hotel-detail**
* ajouter dans le fichier  **index.html** : **<base href= ‘’/’’>** ceci va permettre d’activer le routage avec un **slash( / )**
* dans le **app.component.html** remplacer le **<app-hotel-list>**  par  **<router-outlet>** elle va appeler les pages concernées en fonction de la route
* a la place du **href=’’#’’**  remplacer par la  **directive [routerLink] = [/path]**
* **routerLinkActive="active’’** permet de mettre un lien en surbrillance
* **ActivatedRoute** cette classe qui Permet d'accéder aux informations sur un itinéraire associé à un composant chargé dans un outlet. Elle dispose des attributs, mais nous utiliserons :
  + **snapshot:**  la vue actuelle de la route
  + **paramMap :** Un Observable qui contient une carte des paramètres obligatoires et facultatifs spécifiques à l'itinéraire. La carte permet de récupérer des valeurs uniques ou multiples à partir d'un même paramètre.

**NB :** pour convertir **String -> number** on met juste un **+** devant le string

**Exple :** const **age** = + ‘’12’’ on obtient **age = 12**

**.snapshot =**  c’est pour dire la valeur initiale

**.paramMap.get(‘’id’’)** = ici **id**  doit être le même que celui écrit dans le  **path**  au niveau de  **RouterModule.forRoot()**

**const id: number = this.route.snapshot.params['id'];**  c’est cette methode ci qui fonctionne.

**Exple** : **public hotel:IHotel = <IHotel>{};** défini une variable **hotel** **vide** et de type IHotel

1. **Protéger la navigation avec les Guards(canActivate)**

* **Proj :Continuions avec notre projet**
* Après avoir créé un bouton dans le html de **hotel-detail** et un methode pour faire revenir a la liste des hotels
* Cree une  **variable**  de type **Router** qui va initialiser notre route vers notre page. Ainsi cela va injecter notre service dans se component
* On utilise **Router.navigate([‘url’])**  naviguer sur **url** passe en paramètre. C’est une autre façon a l’instar de

**[routerLink]=’’[‘url’] ‘’** pour naviguer entre les pages.

* **Rôle :**
* Les **Guards**  protègent la navigation( **utilisation des routes**) pendant l’utilisation d’une Application Web.

On distingue :

* **CanActivate**: il surveille l’accès direct à un Url
* **CanDesactivate**: il surveille la navigation en quittant Url.
* **Resolve**: il permet de récupérer certains nombres de données avant de charger Url
* **CanLoad**: il permet de prévenir la navigation asynchrone
* **Création d’un Guard**
* Cree une classe typeScript
* Comme un Guard est un service, alors nous devons l’enregistrer dans le **providedIn :’root’**
* Importer ensuite l’interface du Guard qu’ on dési utiliser
* Implémenter cette interface
* Implémenter les méthodes de cette interface
* Par exemple : **canActivate(ActivatedRouteSnapshot, RouterStateSnapshot)** 
  + **ActivatedRouteSnapshot :** les première valeurs qui vont être capturée lorsqu’ on va atteindre la route
  + **RouterStateSnapshot :** donne l’actuel statut de la route/url qui est en train d’être visitée.
* **Pour protéger notre URL avec le Guard**
* Ne devons repérer dans la liste de nos Url, c’est à dire dans le **app.module.ts** au niveau **RouterModule.forRoot**, url que nous voulons protéger avec notre Guard comme ceci :

**canActivate :[nom\_Guard]**

* Création d un Guard avec Angular CLI :

**ng g g hotel-list/hotel-detail**

* Analyser le resultat de la console apres avoir affiche le contenu de **route** qui est un parametre dans la methode **canActivate(…)** , ainsi on pourra ainsi constate que notre **id**  se trouve dans un **Array** nomme **url.**

1. **Les Modules avec le @NgModule**

C’est une classe TypeScript qu’ on ajoute un décorateur **@NgModule**

Il permet de :

* Grouper les fonctionnalités en bloc et pouvoir réutilise un ensemble de fonctionnalité partout dans notre application
* Organiser les parties de notre applications par exemple en fonctionnalité ou en groupe de fonctionnalité qui partage les dépenses en commun ;
* Préparer l’environnement pour établir les Template plus tard car il met à disposition **des pipes, des composants, et des directives** qui vont être utilisées dans notre module
* Connecter notre application avec des librairies externes
* Ressembler les composants mais aussi de les réexportes pour d’autres modules ;
* **Composition du Module @NgModule**
* **Les imports** : ils nous permettent d’importer les modules Angular par exemple : **HttpClientModule, BrowserModule(**permet l’utilisation de **ngIf et ngFor)**
  + Les modules 3rd Party qui viennent des paquets extérieurs tel que : **Angular-material** ou une autres bibliothèques que nous aimerions utiliser dans notre application.
  + Les custom modules ce sont nos propres modules que nous avons créés
  + Modules pour le routage il est responsable de définir nos routes et de les importées dans notre application
* **Déclaration/bootstrap**  permet de déclarer chaque composant, chaque Directive, chaque Pipe qui doivent être utilise plus tard dans le Template.
* **Exports** il permet d’exporter d’autres modules tel que :
  + Modules Angular
  + Modules 3rd Party
  + Custom Module
  + Component, directive et Pipe
* **Providers**  permet d’enregistrer :
  + Les services
* **Organisation des Modules selon notre projet**
* AppComponent : il utilise le **routerLink** pour cela nous ajoutons le module RouterModule
* HotelListComponent : utilise **ngModele**  pour cela on a besoin du FormsModule on a aussi besoin du BrowserModule a cause de **ngIf** et **ngFor**
* HotelDetailComponent
* HomeComponent

Tout au début du projet on a déclaré notre **StarRatingComponent**  dans le même module que celui de notre  **HotelListComponent**. Mais il est également possible de déclarer le StarRatingComponent dans un autre module et que nous allons importer dans notre module.

Ce module doit tout d’abord **exporter** le StarRatingComponent enfin qu’ il soit vu dans le Module mais aussi dans le HotelListComponent

* **Explication des différent champs du module @NgModule**
* **Bootstrap[…] :**

Chaque module Angule possède au moins un module qui est par convention le **AppModule**  et possède également au moins un composant qui est **AppComponent**.

**AppModule** a travers le **bootstrap[…]** Array va ajouter le **AppComponent**  au processus de démarrage ou le processus de démarrage.

Ce **AppComponent**  va être ajouter dans  **index.html**  à travers son **sélecteur.**

* Utilisation du bootstrap demande  **d’ajouter au moins un composant dans Array de bootstrap[…]** pour avoir une page au démarrage de l’application.
* Seul le module principale doit contenir le **bootstrap[…]**
* **Declaration[…]**

Lorsqu’ un composant est créé il doit être déclaré dans un module avant d’être utilise.

Le module de Array **declaration[…]**  enregistre les **classe typeScript, des composants, des directives et des Pipes.**  Ces éléments peuvent donc être utilisés dans d’autre composants qui se trouvent dans le même module.

* Chaque component, Pipe, directive, ne doit appartenir qu a un seul module
* Déclarer uniquement les composants, Pipes, directives dans **Array declaration[…]** mais jamais  **d’autres classes que nous avons créé nous-même.**
* Jamais redéclaré un composant, Pipe, directive qui a déjà été déclaré dans un module utilisé par le module actuel.
* Chaque composant, Pipe, directive est par défaut prive et ne peut qu’ être utilise dans les composant de même module.
* **Exports[…]**

Avec les exports[…] on peut exporter les **modules internes d Angular** tel que les **FormsModule,** enfin que ceci soit visible à travers  **d’autres modules**  qui utiliserons nos modules.

* Un module peut exporter les composants, Pipes, directives pour d’autres composants
* Il peut réexporter un module pour exporter ses composants, directives et Pipes
* On peut  **réexporter un module sans avoir besoin de l’importer**

Par exemple si on utilise les **SharedModule**  auquel on veut exposer les **FormsModule** on n’est pas oblige au préalable  **d’importer les FormsModule.** On ne peut cependant directement  **exporter les FormsModule** et ainsi, ils seront disponibles dans chaque composant qui va **importer le SharedModule**  pour chaque Module du moins qui importera ce SharedModule

* **Ne jamais utiliser Array exports[…] pour exporter un service**
* **Imports[…]**
* Un module importé met a disposition toute directive, Pipe, component qui sont exportes par ce module.
* Toujours importer uniquement ce dont nous avons besoin
* Un module importe ne donne pas accès aux modules qui y sont importés, mais seulement ceux exportés

Imaginons que nous ayons **AppModule** qui déclaré **HotelListComponent** et ainsi on peut afficher HotelListComponent a nos utilisateurs.

Si on veut utiliser le **StarRatingComponent,**  on doit créer un module par exemple  **SharedModule**. Ce SharedModule doit déclarer le StarRatingComponent et nous allons **importer** le SharedModule dans notre **AppModule**. Mais le StarRatingComponent n’est pas encore visible pour notre HotelListComponent. Pour **résoudre** cela, nous devons  **exporter**  dans notre SharedModule le StarRatingComponent.

Imaginons maintenant que nous voudrions utiliser le **ngModel** dans toute notre application, mais nous voulons l’ajouter à notre SharedModule.

**Si**  on importe seulement le **FormsModule**  dans notre SharedModule, cela va ajouter le **ngModel** pour le StarRatingComponent mais pas pour **HotelListComponent**

**Que faire pour avoir le ngModel dans notre HotelListComponent ?**

Il faudra que le SharedModule puisse **exporter**  le FormsModule en plus de l’avoir  **importer**, ainsi le ngModel sera disponible pour le StarRatingComponent et HotelListComponent.

* **Providers[…]**

**Ancienne façons d’injecter les services**

**Ce dernier ne sera disponible que dans**

**Son module**

Nouvelle Façons

**Service sera disponible partout dans l’application.**

* Tout service qui est enregistre dans Array des providers[…] est ajoute au root injector et disponible dans toute l’application

**Attention !!!**

Si on crée un sous module **HotelModule** différent du module principale, et à l’intérieur on introduire un Array **providers[…],**  on ajoute par exemple **HotelService**  dans cet Array là. HotelService ne sera pas visible seulement dans HotelModule, mais il sera visible dans toute l’application parce que le **providers[…]**  ajoute cela au **root injector. Nous devons faire attention à utilise les services dans Array des providers**

* Ne jamais ajouter des services au providers[…] du SharedModule

1. **Structure l’application en module**

* **Proj :Continuions avec notre Proj**
* Cree une dossier **hotels** à l’intérieur de **app**
* Cree dans le dossier hotels le dossier **hotel-list**  et **hotel-detail**
* Déplacer les fichiers **.ts , .html et .css** de hotel-list et hotel-detail dans les nouveaux dossiers crées respectifs
* Créer un dossier **shared** a l’intérieur de hotels
* Dans shared crée les dossiers : **models, services, guards**
* Déplaçons maintenant **hotel.ts**(qui est notre interface) dans les **models, .service.ts** dans les **services**  et les **.guard.ts** dans les **guards.**
* On doit maintenant modifier les références dans le fichier **app.module.ts**
* Cree un nouveau module : **ng g m hotels/hotel --flat -m app**
  + **/hotel :**  est le m´nom de notre module
  + **--flat :** signifie qu’on ne crée pas de sous dossier
  + **-m :**  pour qu’ il importe ça dans le module principale(**appModule**)
  + **app:** pour qu’il retrouve **app.module.ts**
* configure notre **hotel.module.ts**
* supprimer l’import de nos composant **HotelList et HotelDetail** dans le module app.module.ts, car nos composants n’ont pas le droit d’être dans 02 modules à la fois
* on va importer le **Router.forChild()**  car on a déjà le **.forRoot()** dans le app.module.ts
* créer un nouveau module :

**ng g m shared/shared --flat -m hotels/hotel.module**

* **-m hotels/hotel.module:**  c’est pour qu’ il soit importé dans le **hotel.module**
* Exporter **commonModule, FormsModule, StarRating et le ReplaceComma**  dans le shared.module

**Fig. Structure de notre Application**

* **Création d’un module pour nos route**
* Création d’un module pour nos route pour le module **hotel.module et app.module :**

**ng g m hotels/hotel-routing --flat**

* **Exporte**  le RouterModule dans notre **hotel-routing.module**
* faire la même chose pour le app.module

**ng g m app-routing --flat**

* en faisant cette opération pour le **app.module**  il y a une erreur qui est générée, car on pourra plus accéder a nos route, pour cela **il faut importer AppRoutingModule**  **en dernier position.** C’est la cause de **{path:"\*\*", redirectTo:"home", pathMatch:"full"}**  qui bloque les autres routes, voilà pourquoi il doit être en dernier position.

1. **Rendre actif un lien enfant avec : [routerLinkActiveOptions]**

* **Proj :Continuions avec notre Proj**
* Générer un nouveau composant pour nos formulaire :

**ng g c hotels/hotel-edit**

* Créer une route qui va rediriger vers se composant
* **[routerLinkActiveOptions]="{exact:true}"** ceci permet de rendre un lien en surbrillance et de laisser las autres san ça. On doit le mettre sur la **route enfant et celle du parent**

1. **Construction du Template de formulaire réactif**

* **Proj :Continuions avec notre Proj**
* On va définir le **hotel-edit.component.html**
* Aller dans le **SharedModule**  et exporte le **ReactiveFormsModule**
* Créer une variable de type **FormBuilder** dans le constructeur du fichier **hotel-edit.component.ts**
* Allez dans le hotel-edit.component.html et désactivé le **validation** en faisant **novalidate**
* Connecter notre formulaire au FormGroup avec **[formGroup]=’’hotelForm’’**
* **(ngSubmit)=’’methode()’’ :** permet de que lorsqu’ on appuis sur le bouton **submit** la methode **methode()**  soit appelée.
* On peut récupérer les données grâces aux paramètres de **URL**  avec **ActivatedRoute**
  + **1-**  importer ActivatedRoute
  + **2-** inscrit une variable de type **ActivatedRoute** dans le constructeur(**injecteur de dépendance**)

**constructor(private route: ActivatedRoute){…}**

* + **3-** Ensuite on applique la methode suivante :
* La methode **FormGroup.patchValue({…})** permet d’attribuer les valeurs à notre formulaire
* La methode **of(objet)**  permet de créer un observable à partir de l’objet.
* **Prévenir les pertes de données avec les Guards**
* Créer un nouveau fichier hotel-edit à l’intérieur du dossier guards
* sélectionner **CanDeactivate**
* la propriété **dirty**  de l’objet  **FormGroup** (**FormGroup.dirty**), permet de savoir si un objet contient quelque chose.
* Ajouter ce guard dans le **hotel-routing.module.ts**

Cette expression signifie que on récupère la valeur du champs **hotelName**  dans notre formulaire et si cette valeur n’existe pas alors on retourne **nouveau hotel**

*const* hotelName = component.hotelForm.get("hotelName").value || 'nouveau hotel';

1. **Intégrer In-Memory-API et faire appel aux données**

* Installation de Angular In-memory-web  se place sur la racine du dossier mon-app:

**npm i angular-in-memory-web-api --save-dev**( **nicht geht !!!** )

**npm i angular-in-memory-web-api@0.13.0 --save-dev**

* Créer un nouveau dossier **api**  dans le dossier **shared**
* Dans api crée un fichier **hotel.data.ts** qui va simuler notre nouvelle base de données.
* Cree une classe dans hotel.data.ts et qui implémente le service : **InMemoryDbService**
* Importer dans hotel.module.ts **InMemoryWebApiModule.forFeature(nom de la classe)**

1. **Ajouter un nouvel élément avec http.post()**

Après avoir créé la nouvelle methode **createHotel(hotel :IHotel)**

* On doit maintenant aller l appelé dans la methode **saveHotel()**  du fichier **hotel-edit.component.ts**  quand **hotel.id =0.**
* Se rassure que les noms des éléments figurant **FormGroup** figure bien dans notre base de données.
* Le nouvel hotel qu’ on crée s affiche sans image et son id=0, pour cela dans la methode **createHotel** () on va ajouter une **image par default** et **id=null** :
* Ensuite il faut faudrait que nos id s incrémenté lorsqu’ un nouvel hotel est créé : c’est grâce à la methode **genId() de InMemoryDbService** dans le fichier **hotel.data.ts**

genId(*hotels*: IHotel[]):*number*{

        return *hotels*.length>0? Math.max(...*hotels*.map(*hotel=>hotel*.id))+1: 1;

    }

Cette methode prend en paramètre une liste d hotel, on teste si y a des éléments dans la liste si c’est vrai alors on appel la fonction **Math.max()** pour trouver la valeur la plus grande. On décompose la liste d hotel ensuite on appel la fonction  **map()** sur chaque **hotel :IHotel** . Sur chaque hotel on récupère leur id, qu’ on compare pour trouver le maximum, c’est ainsi qu’ on ajoute **1**  a chaque fois. Mais si tout ceci n’est pas possible alors on retourne 1 comme valeur de **id**

1. **Supprimer des éléments avec http.delete()**

* Créer une nouvelle methode dans le service : **deleteHotel(id :number)**

public deleteHotel(*id*:*number*): Observable<{}> {

*const* url = `${this.HOTEL\_API\_URL}/${*id*}`;

  return this.http.delete<IHotel>(url).pipe(

    catchError(this.handleError),

  )

}

* On se rend dans la **hotel-edit.component.ts** pour créer une methode pour supprimer les hotels.

public deleteHotel():*void*{

    if(confirm(`Are you sure you want to delete ${this.hotel.hotelName}`)){

      this.hotelService.deleteHotel(this.hotel.id).subscribe({

        next: ()*=>* this.saveCompleted(),

      })

    }

  }

* Ensuite le connecte avec le **Event binding sur le fichier html**  du bouton supprimer.
* Ensuite faut rendre le bouton supprime inactif lorsqu’ on veut ajouter un nouveau hotel :

[disabled]="hotel.id === 0"

1. **Ajouter et supprimer dynamiquement les champs dans un formulaire**

**But :**  d ajouter plusieurs étiquettes à notre formulaire

* Ajouter un autre champ juste après Evaluation dans **hotel-edit.component.html**
* Tout d’abord ajouter une ligne

<div class="form-group row mb-2">

à la suite **d’Evaluation.**

* Ajouter dans le **hotelForm** un attribut: **tags** : qui est un **Array vide**

tags: this.fb.array([])

* Créer une methode qui va prendre la référence de cet Array tags

public *get* tags(): FormArray{

    return this.hotelForm.get('tags') as FormArray;

  }

* Créer maintenant une nouvelle methode qui va ajouter les éléments dans ce Array :

public addTags():*void*{

    this.tags.push(new FormControl())

  }

* Ajouter un clic qui prendra cette methode **addTags()** sur notre bouton  **Ajout d’étiquette.**
* Créer une **div formArrayName=’’tags’’** et inséré tous nos champs sur les étiquettes à l’intérieur.
* Sur cette <div> on fait une itération avec  **\*ngFor** et on récupère les index pour stocker dans une variable avec :  **let i = index ;**

<div formArrayName="tags" \*ngFor="*let* tag of tags.controls; *let* i=index;">

* On se sert du Property binding pour donne une attribut **for** a chacun de nos label :

[attr.for]="i"

* Ajouter maintenant une methode pour supprimer nos étiquettes créées . La methode **FormArray.markAsDirty()** permet de dire au formulaire globale qu’ un élément a été retiré de la liste

 public deleteTag(*index*: *number*){

    this.tags.removeAt(*index*); // supprime un index

    this.tags.markAsDirty();

  }

* Maintenant on ajoute cette methode a notre bouton supprimer.
* Ajouter un nouvel attribut **tags ? : String[]**  dans **hotel.ts**
* Aller dans notre **hotel-liste.component.html** et modifier nos badge. Et devient ainsi :

<div \*ngFor="*let* tag of hotel.tags">

     <div \*ngIf="showBadge">

        <span class="badge rounded-pill bg-info" >{{tag}}</span>

     </div>

</div>

* On voudrait passer la valeur de notre Array **tags** a notre formulaire a l’aide de la methode **patchValue({…})**  comme on a fait pour les autres éléments. Mais cela se fait autrement avec  **setControl()** :

this.hotelForm.setControl('tags', this.fb.array(this.hotel.tags || []));

* Quand on a affaire aux directives **\*ngIf et \*ngFor**  , il faudrait utiliser **<ng-container></ ng-container>** a la place des **div**. Ces balises n apparaitrons pas dans le html.

  <ng-container \*ngFor="*let* tag of hotel.tags">

     <ng-container \*ngIf="showBadge">

        <span class="badge rounded-pill bg-info" >{{tag}}</span>

     </ng-container>

  </ng-container>

* **[ngClass]=’’…’’** permet d’ajouter une classe en fonction d’une condition donnée.

<ng-container \*ngFor="*let* tag of hotel.tags; *let* i= index">

   <ng-container \*ngIf="showBadge">

      <span class="badge rounded-pill"

      [ngClass]="i%2===0? 'bg-info': 'bg-success'">{{tag}}</span>

  </ng-container>

</ng-container>

1. **Intercepter les erreurs de serveur et les affichées**

* Créer une nouvelle propriété **erroMessage** dans la classe du fichier **hotel-edit.component.ts**
* Ensuite on modifie la methode saveHotel() :

* Ensuite on ajoute ce message dans notre template sous forme d’une alerte

1. **Tutoriel sur RxJS**

**RxJS**: **R**eactive E**x**tension for **J**ava**S**cript

C’est une bibliothèque JavaScript pour la composition de programmes Asynchrones et base sur des évènement en utilisant des séquences d observables.

* Installation de Angular/cli dans le dossier de ton projet :

**npm install -g @angular/cli**

* Cree un nouveau projet : **ng new rxjs-guru --defaults –skip-git**
* Lancer Angular : **npm run start** oder **ng serve --open**

1. **Gérer un Observer et un Subscriber**

**Un Observer(objet javaScript)**  est une collection de fonction de rappel qui savent à quel moment écouter les valeurs qui doivent être délivrée à travers un observable.

1. **Observable**

**Rmq :** Pour notre Exemple : C’est l’ensemble des nouvelles boites qui arrivent sur la bande.

**Un Observable** ou **séquence d observable**  peuvent être des objets ou des éléments de différents types. Ces types la peuvent être :

* **String,**
* **number,**
* **Event(**keydown, keyup, …)
* **des objets** composes de différents types de variables et de propriétés également.
* **Une réponse d’une requête http** envoyé et que réponse est utilisée dans notre application. **Une combinaison d’autres Observables** qui arrive à différent moment. **Un observable n’est pas toujours Asynchrone**
* **Methode1 : Création d’un Observer**
* Appeler le constructeur d observable avec RxJS :

**new Observable(Observer)**

* Ensuite démarrer la séquence en appelant un **subscribe(observer\_cree) :**

stream.subscribe(observer);

* **Methode2 : Création d’un Observer**
* On peut le faire directement dans la methode **subscribe()** sans passer par la création d’un Observer au préalable.

*const* subscription = stream.subscribe(

*item* *=>* console.log(`Une boite arrive ${*item*}`),

*err* *=>* console.log(`OUps il y a eu une erreur ${*err*}`),

      ()*=>* console.log('Terminer plus rien')

    );

* **Comment Stopper une Séquence d observable**

*myObserver*.complete();

* + 1- En se servant de

*const* stream = new Observable(*myObserver* *=>*{

*myObserver*.next('boite1');

*myObserver*.next('boite2');

*myObserver*.complete();

*myObserver*.next('boite3');

    });

La **complete()**  vient avant **boite3** dont se dernier ne sera pas émis

* + 2- En introduisant une erreur avant la fin de la séquence :

Ici on introduit une **Error** avant la dernier boite donc **boite3**  ne sera pas émis

*const* stream = new Observable(*myObserver* *=>*{

*myObserver*.next('boite1');

*myObserver*.next('boite2');

*myObserver*.error(new *Error*());

*myObserver*.next('boite3');

*myObserver*.complete();

    });

**Résume :**