IV. Création d'une application festival

Installation et création d'une appli Angular

Stackblitz

Pour ceux travaillant avec Stackblitz, créer une nouvelle application sur le site

Installation locale

`Prérequis :` Node.js doit être installé sur la machine

Il faut commencer par installer, si ce n'est déjà fait, l'interface ligne de commande de angular :

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

Une nouvelle application se crée par la commande suivante (un répertoire se créé) :

```
ng new FestivalApp
```

Validez les 3 réponses par défaut (par la suite, l'option Router pourra être activée)

Mise en place du squelette de l'application

Générez la classe métier de base :

-> création d'un répertoire model, puis la classe festival à l'intérieur

```
ng g class models/festival
```

-> création d'un composant festivals-list dans un répertoire festival, lui-même dans un répertoire composants

```
ng g component components/festival/festivals-list
```

-> création d'un composant festival-details

```
ng g c components/festival/festival-details
```

Mise en place du squelette de l'application (suite)

-> création d'un service festivals

```
ng g service services/festivals
```

-> création d'un service festivaljson

```
ng g s services/festivaljson
```

-> création d'un service message

```
ng g s services/message
```

-> création d'un composant message qui servira à afficher les messages du service message

```
ng g c components/shared/message
```

Lancez le serveur et visualisez l'application à l'URL http://localhost:4200/

```
ng serve
```

Class Festival : 1ère version


```
export class Festival {
   static sqmTable = 6;
   public id?: string;
   public name!: string;
   public tablemax_1: number;
   public tableprice_1: number;
   public sqmprice_1: number;
   public tablebooked_1: number = 0;
   public sqmbooked_1: number = 0;
   public tablemax_2: number;
   public tableprice_2: number;
   public sqmprice_2: number;
   public tablebooked_2: number = 0;
   public sqmbooked_2: number = 0;
   public tablemax_3: number;
   public tableprice_3: number;
   public sqmprice_3: number;
   public tablebooked_3: number = 0;
   public sqmbooked_3: number = 0;
   public revenue: number = 0;
   public visitor: boolean = false;
   public get tableTotal() : number { return this.tablemax_1 + this.tablemax_2 +
this.tablemax_3; }
}
```

Paramètres optionnels des constructeurs

L'objectif est d'ajouter un constructeur à la classe Festival permettant de créer un festival.

On souhaite pouvoir créer un Festival avec juste un nom, seul paramètre obligatoire donc. Pour cela on pourra utiliser la directive `@Optional()` qui indique qu'un paramètre du constructeur peut être ignoré (attention l'ordre est important).

Pour utiliser cette directive, il faut l'importer :

```
import { Optional } from "@angular/core";
```

Et écrire les paramètres sous la forme :

```
@Optional() sqmprix_1: number?;
@Optional() tablereserv_1: number = 0;
```

et dans le code du constructeur :

```
this.sqmprice_1 = (sqmprix_1 == null) : this.tableprice_1/6 : sqmprix_1;
this.tablebooked_1 = tablereserv_1;
```

Exercice 2 : Faite un fork de votre projet avant de continuer

Modifiez la classe métier Festival en respectant les consignes ci-dessus.

Initialisez un tableau de plusieurs festivals dans le composant principal

Affichez le composant festivals-list dans le composant principal

Faites en sorte que le composant festivals-list affiche la liste des festivals initialisée dans le composant principal pour obtenir quelque chose comme ci-dessous:

festivals-list works!

Id	Nom	tables	Prix Espace entrée	Prix Espace salle	Prix Espace buvette	m^2 entrée	m^2 salle	m^2 buvette	# tables entrée	# tables salle	# tables buvette
	Fjm 2018	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040
	Fjm 2019	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040
	Fjm 2020	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040

end of table

V. Dependency Injection

Les services : principe

Un service est généralement une classe dont l'objectif est précis et peut être utile à plusieurs composants.

En séparant la gestion du composant des autres types de traitement, les composants sont plus efficaces et concentrés dans leur rôle initial. Cela permet d'éviter le syndrome du fat controller.

Les tâches indépendantes de l'expérience utilisateur, comme l'extraction de données du serveur, la validation des entrées de l'utilisateur ou l'affichage de message de log, seront délégués à des services

Les services

Afin d'éviter que chaque composant crée sa propre instance d'un service, et garantir des traitements cohérents (éviter les collisions, garantir une source de vérité unique, ...), on préfèrera injecter une dépendance vers une instance précise d'un service.

Exemple :

```
export class MessageComponent implements OnInit {
  constructor(public messageService : MessageService) { }
  ngOnInit(): void {
  }
}
```

Ici on injecte le `MessageService` dans le composant `MessageComponent`.

Reste à déterminer qui crée l'instance partagée de MessageService et qui gère son cycle de vie -> ceci est fait par le mécanisme de DI* de Angular

* *pour Dependency Injection*

L'Injection de dépendance dans Angular

L'injection de dépendance met en œuvre cinq acteurs :

Consommateur

La classe qui utilise l'instance injecté, généralement le composant utilisant un service

Dépendance

L'instance injectée, généralement le service

DI Token

Un token qui identifie une dépendance. On utilise un tel token quand on déclare une dépendance -> géré par Angular

Fournisseur

Le Fournisseur indique à un injecteur comment obtenir ou créer une dépendance.

Injecteur

L'injecteur crée des dépendances et maintient un conteneur d'instances de dépendances qu'il réutilise si possible.

L'injecteur est donc le mécanisme principal. Un injecteur est créé pour l'application et d'autres injecteurs peuvent être créés si besoin. C'est angular qui s'en charge.

Pour cela, pour toute dépendance, il faut enregistrer un fournisseur auprès de l'injecteur de l'application. Cela peut se faire de deux manières différentes :

1. via la directive *@Injectable()* en indiquant entre les parenthèse qui est le fournisseur

```
@Injectable({
   providedIn: 'root' // la racine de l'application est un fournisseur
})
```

Note : `@Injectable()` sert également à indiquer qu'une classe est injectable. Elle n'est pas nécessaire pour les `@components`, `@pipe` ou `@directive` qui sont injectables par défaut.

2. En indiquant dans la directive du module `@NgModule` ou dans celle d'un composant `@Component` un tableau de fournisseurs :

```
providers: [ MessageService ]
```

qui est un raccourci pour

```
providers: [ { provide: MessageService, useClass:
MessageService} ]
```

3. En créant un fournisseur :

```
export const MessageProvider = [
  { provide: MessageService, useClass: MessageService, multi:
false }
];
```

Retour sur les services et l'injection

Il faut donc enregistrer au moins un fournisseur pour le service que l'on veut utiliser.

Par défaut, la commande `ng generate service` enregistre un fournisseur avec l'injecteur racine :

```
@Injectable({
 providedIn: 'root',
})
```

Angular crée alors une instance unique et partagée du service et l'injecte dans toute classe qui le demande.

Exercice 3

Implémentez le MessageService :

Celui-ci maintient une liste de messages et possède deux fonctions :

`log(string)` qui ajoute un message à la liste de messages

`clear()` qui efface la liste (réinitialise la liste)

Implémentez le composant MessageComponent :

Celui-ci affiche la liste des messages du service et un bouton clear qui efface la liste.

Ajoutez le composant à l'affichage du composant principal sous la liste de festivals et faite en sorte que le composant FestivalList affiche un message lors de son initialisation.

Celui-ci doit s'afficher dans la page du composant principal sous la liste

festivals-list works!

Id	Nom	tables	Prix Espace entrée	Prix Espace salle	Prix Espace buvette	m^2 entrée	m^2 salle	m^2 buvette	# tables entrée	# tables salle	# tables buvette
	Fjm 2018	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040
	Fjm 2019	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040
	Fjm 2020	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040

end of table

Messages

clear

Affichage de la liste de festivals - 3 festivals

VI. Les Formulaires

Forms : introduction

Angular propose deux approches différentes pour les formulaires

1. `réactive`
2. pilotée par un `modèle`.

Les deux ont des principes communs :

- capture de la saisie à partir de la vue
- validation de la saisie
- création d'un modèle de formulaire et d'un modèle de données
- suivi des modifications.

1. les formulaires réactifs fournissent un accès direct et explicite au modèle du formulaire. Ils sont plus robustes, plus évolutifs, plus réutilisables.
   Ils sont à privilégier si les formulaires sont un élément clé.
2. les formulaires pilotés par des modèles s'appuient sur les directives du modèle pour créer et manipuler le modèle de données sous-jacent.
   Ils sont faciles à utiliser pour ajouter un formulaire simple à une application.
   Ils sont à utiliser si les formulaires sont très simples et que la logique peut être gérée uniquement dans le modèle du formulaire.

Mise en place d'un formulaire

Formulaire piloté par le modèle

Dans le html

```
<input type="text" [(ngModel)]="festival.name">
```

et dans la classe :

```
@Input() festival: Festival;
```

Toute modification de la propriété `name` de festival entraîne le changement de la vue, toute saisie dans le input provoque le changement dans la propriété de festival

Formulaire réactif

Dans le module, importez le `ReactiveFormsModule` et rajoutez le dans la liste des import

```
import { ReactiveFormsModule } from '@angular/forms';
imports: [ BrowserModule, ReactiveFormsModule],
```

Dans le code du composant

```
import { FormControl } from '@angular/forms';
```

puis création d'une instance de FormControl

```
@Input() festival: Festival;
nameControl : FormControl;
ngOnInit(): void {
 this.nameControl = new FormControl(this.festival.name);
}
```

Dans le html

```
<input type="text" [formControl]="nameControl">
```

Cette fois-ci les données sont échangées entre la vue et le FormControl, festival.name n'est pas affecté. Il faudra reporter la valeur une fois le formulaire validé

Flot des données de la vue vers le model

- L'utilisateur tape une valeur dans l'élément de saisie
- L'élément de saisie du formulaire émet un événement "input" avec la dernière valeur.
- L'accesseur qui écoute les événements sur l'élément d'entrée du formulaire transmet immédiatement la nouvelle valeur à l'instance de `FormControl`.
- L'instance du `FormControl` émet la nouvelle valeur via l'observable `valueChanges`.
- Tous les abonnés à l'observable `valueChanges` reçoivent la nouvelle valeur.

On peut accéder à la valeur grâce à la propriété `value` du `FormControl` pour affecter cette valeur à `festival.name`

On peut aussi observer les changements grâce à la propriété observable `valueChanges`

```
this.nameControl.valueChanges.subscribe(
   (name) => { this.festival.name = name; }
);
```

Exercice 4

Implémentez le composant FestivalDetails

Ce composant aura en `@Input()` un festival et affichera un <input> permettant d'éditer ce nom. On rajoutera un bouton de validation qui, quand on le cliquera, affectera la valeur du FormControl au nom du festival.

Dans le composant principal, on affichera sous la liste, avant les messages, le composant FestivalDetails en lui passant en paramètre le premier festival de la liste.

À chaque validation, le nom du premier festival de la liste doit se mettre à jour.

Bonus :

implémentez le suivi de valueChange pour voir les changements en direct, revenant à faire un formulaire template drive

Formulaires groupés

Un formulaire contient généralement plusieurs éléments de saisie.

Les formulaires réactifs permettent de regrouper plus contrôles de deux manières :

1. Un groupe de formulaires
2. Un tableau de formulaires permettant de définir un formulaire dynamique.

On abordera ici les groupes de formulaires.

Pour les formulaires dynamique, plus complexes à mettre en œuvre, mais aussi moins utilisés, se référer à la documentation officielle.

Pour utiliser les groupes de formulaire, vous devez importer la classe FormGroup dans votre composant :

```
import { FormGroup, FormControl } from '@angular/forms';
```

Puis définir un groupe de FormControl

```
this.festivalGroup = new FormGroup({
   name: new FormControl(this.festival.name),
   entrancePrice: new FormControl(this.festival.tableprice_1)
});
```

Puis ajouter le formulaire dans le code html de votre composant :

```
<form [formGroup]="festivalGroup">
  <label>Name: <input type="text" formControlName="name"></label>
  <label>Entrance: <input type="text"
formControlName="entrancePrice"></label>
</form>
```

Le formulaire groupé permet de récupérer toutes les valeurs d'un coup.

Pour cela on rajoutera l'ajout d'une fonction de validation via l'évènement `ngSubmit` :

```
<form [formGroup]="festivalGroup" (ngSubmit)="validate()">
```

L'évènement `ngSubmit` du formulaire et lié à l'évènement `(click)` d'un bouton de type submit. Il suffit donc de rajouter un tel bouton pour valider le formulaire en une fois :

```
<button type="submit" [disabled]="!festivalGroup.valid">Submit</button>
```

`festivalGroup.valid` garantit qu'on ne pourra pas valider si la saisie dans les éléments input n'est pas valide. En effet, un formulaire permet d'indiquer des conditions de validation de la saisie

la méthode `get(nomControl).value` permet de récupérer la valeur d'un FormControl du FormGroup

```
this.festival.name = this.festivalGroup.get('name').value;
```

la méthode `setValue()` permet de mettre à jour le formulaire entier, la méthode `patchValue()` permet de mettre à jour certaines valeurs du formulaire :

```
this.festivalGroup.patchValue({name:"toto"});
```

FormBuilder

Créer un formulaire groupé est assez répétitif et fastidieux. Heureusement Angular fournit un service FormBuilder permettant de faciliter la création des formulaires groupés

```
import { FormBuilder } from '@angular/forms';
```

puis injectez le service dans votre composant, nommons le fb

Et enfin créez votre formulaire :

```
this.festivalGroup = this.fb.group({
  name: [this.festival.name],
  entrancePrice: [this.festival.tableprice_1]
});
```

La méthode pour accéder aux formControl du group est `get(nomducontrol)`. Ainsi

`this.festivalGroup.get('name').value` permet de récupérer la valeur saisie pour le nom.

Validation des saisies

Angular fournit une classe de validation de saisie et permet également de créer ses propres validateurs.

Pour utiliser le validateur fournit par angular, il faut d'abord importer la classe :

```
import { Validators } from '@angular/forms';
```

Puis rajouter le type de validation qu'on souhaite, par exemple :

```
this.festivalGroup = this.fb.group({
  name: [this.festival.name, [Validators.required,
  Validators.minLength(4)]],
  entrancePrice: [this.festival.tableprice_1, [Validators.required,
  Validators.min(80)]
  ]
});
```

Validateur custom, exemple :

```
import { AbstractControl, ValidationErrors } from '@angular/forms';

export class UsernameValidator {
    static cannotStartOrEndWithSpace(control: AbstractControl) :
ValidationErrors | null {
        if (control.value.startsWith(' ')) { return
{cannotStartOrEndWithSpace: true}; }
        if (control.value.endsWith(' '))   { return
{cannotStartOrEndWithSpace: true}; }
        return null;
    }
    static cannotContainSpace(control: AbstractControl) :
ValidationErrors | null {
        if((control.value as string).indexOf(' ') >= 0){
            return {cannotContainSpace: true}
        }
        return null;
    }
}
```

Exercice 5

Formulaires

1. Modifiez votre composant FestivalDetails pour intégrer un formulaire groupé permettant de changer au moins 3 valeurs du Festival.
2. Ajoutez un bouton à chaque ligne de votre liste de festival qui permet de sélection un festival et faite en sorte que soit affiché non plus le premier festival, mais le festival sélectionné.

Bien sûr, si aucun festival n'est sélectionné, le composant FestivalDetails ne doit pas s'afficher.

festivals-list works!

Id	Nom	tables	Prix Espace entrée	Prix Espace salle	Prix Espace buvette	m^2 entrée	m^2 salle	m^2 buvette	# tables entrée	# tables salle	# tables buvette	
	Fjm 2018	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040	Select
	Fjm 2019	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040	Select
	Fjm 2020	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040	Select

end of table

Messages

clear

Affichage de la liste de festivals - 3 festivals

festivals-list works!

Id	Nom	tables	Prix Espace entrée	Prix Espace salle	Prix Espace buvette	m^2 entrée	m^2 salle	m^2 buvette	# tables entrée	# tables salle	# tables buvette	
	Fjm 2018	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040	Select
	Fjm 2019	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040	Select
	Fjm 2020	176	$110.00	$100.00	$90.00	$18.30	$16.70	$15.00	064	072	040	Select

end of table

festival-details works!

Name: Fjm 2018

Entrance: 110 Room: 100

Validate change

Messages

clear

Affichage de la liste de festivals - 3 festivals

hello

{"tablebooked_1":0,"sqmbooked_1":0,"tablebooked_2":0,"sqmbooked_2":0,"tablebooked_3":0,"sqmbooked_3":0,"revenue":0,"visitor":false
2018","tablemax_1":64,"tablemax_2":72,"tablemax_3":40,"tableprice_1":110,"tableprice_2":100,"tableprice_3":90,"sqmprice_1":18.3,"sqmp