Angular

Angular kesako ?

Angular est un framework développé par Google pour le développement Web

Attention : Angular ≠ AngularJs

Angular est construit sur le langage TypeScript et se base sur

- des composants pour la création d'applications web évolutives.
- des bibliothèques qui couvrent une grande variété de fonctionnalités, notamment le routage, la gestion des formulaires, la communication client-serveur, etc.

Angular : les points clefs

Angular s'inspire du design pattern MVC mais en y intégrant des aspects programmations réactives -> MVI ?

Les concepts principaux mise en œuvre sont :

- des *composants* qui se décomposent en :
  - une classe qui décrit la logique -> sorte d'intent
  - une vue composée de :
    - un *html* étendu avec une nouvelle syntaxe (des *templates*) permettant d'intégrer des aspects *react* et même des instructions
    - un *style* pour le look (css, scss ou sass)

- des *directives* qui ajoutent des fonctionnalités aux éléments (.ts, .html) des applications Angular.
- l'*injection* de dépendance
- la navigation intra via une librairie de *Routing*
- la navigation
  - externe via un module HTTP
  - ou via des librairies propres à des fournisseurs (Firebase)
- des formulaires (*Form*) intégrant des éléments de contrôles liés aux propriétés et comportant des validateurs

Les composants

Les composants : résumé

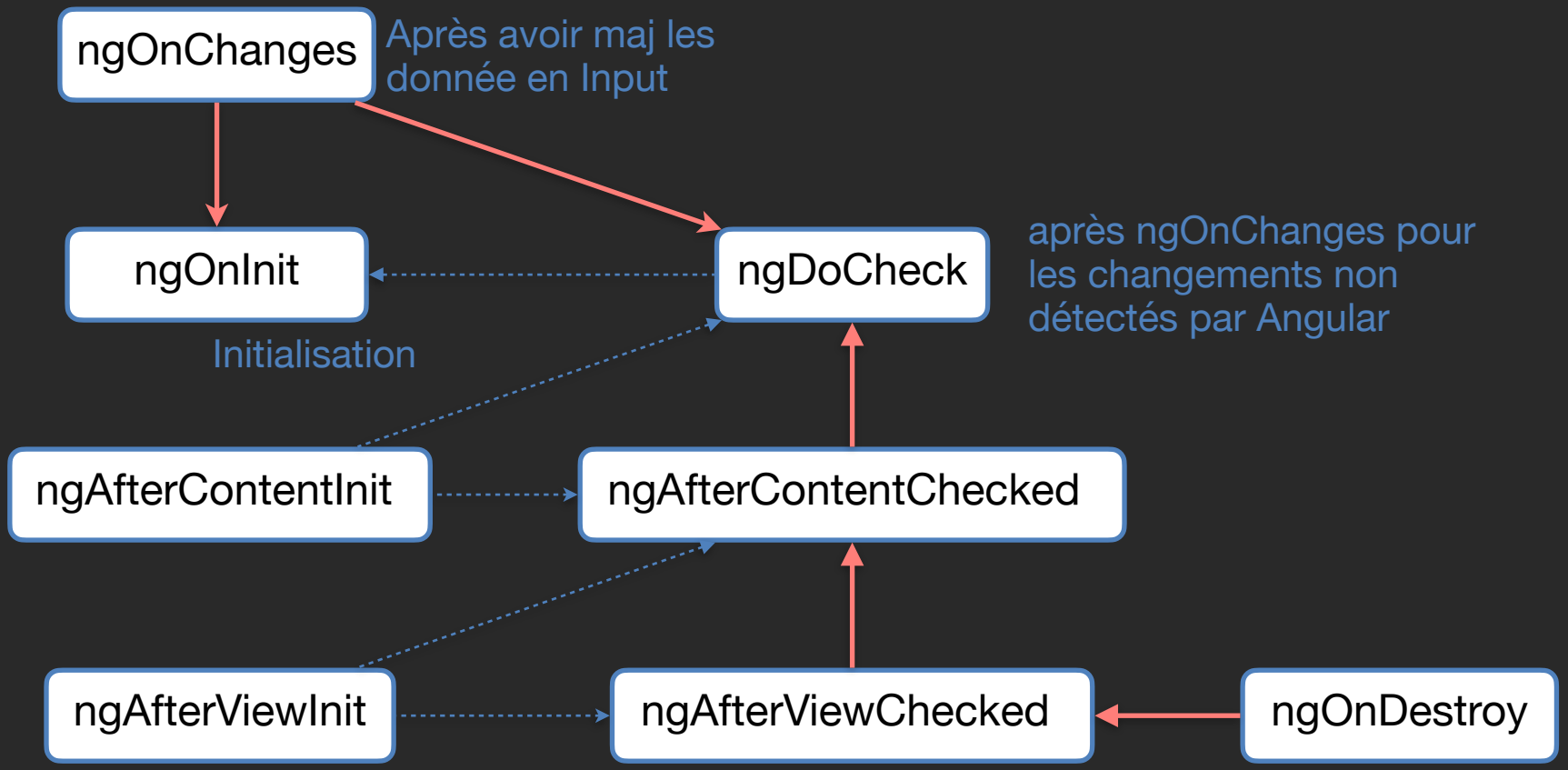
Ouvrez Stackbliz et créer un nouveau projet Angular :

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

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})
export class AppComponent  {
  name = 'Angular ' + VERSION.major;
}
```

- `selector` : sélecteur qui permettra d'insérer le composant dans une page html
- `templateUrl` : la vue décrite en html + directives

Cycle de vie


ngOnChanges
Après avoir maj les donnée en Input
ngOnInit
Initialisation
ngDoCheck
après ngOnChanges pour les changements non détectés par Angular
ngAfterContentInit
ngAfterContentChecked
ngAfterViewInit
ngAfterViewChecked
ngOnDestroy

L'initialisation d'un composant se fait essentiellement dans la méthode `ngOnInit()`

Toutes les initialisations complexes, y compris la récupération de données doivent se faire ici

Par exemple, on peut modifier le code du composant principal :

```
export class AppComponent {
  name = "Angular " + VERSION.major;
  public festivals;

  ngOnInit(): void {
    this.festivals = [
      { name: "FJM2020", tables: 160 },
      { name: "FJM2018", tables: 80 },
      { name: "FJM2019", tables: 110 }
    ];
  }
}
```

Communication de données entre composants

Il est possible d'échanger des données entre composants :

- données échangées entre une vue parent et un composant enfant : `@Input()`
- interception d'une donnée entrante pour la modifier avant de l'utiliser
- interception d'une donnée entrante avec `ngOnChange()`
- une vue parent à l'écoute d'un évènement d'un composant enfant : `@Ouput` et `EventEmitter`
- interaction d'un parent avec un composant enfant via une variable locale
- injection du composant enfant dans la vue parent pour un accès complet : `@ViewChild()`
- communication entre parent et enfant via un service

Donnée échangée entre parent et enfant : `@Input()`

La directive @Input placé devant le nom d'une propriété indique cette propriété sera passée en "paramètre" au composant par la vue parent.

-> hello.component.ts

```
export class HelloComponent  {
  @Input() name!: string; // s'attend à recevoir une valeur
}
```

Dans app.component.html

```
<hello [name]="name"></hello>
```

Trois indications ici :

- `<hello></hello>` est le `selector` du composant hello : il indique qu'il faut l'afficher ici

- `name =` indique une variable `@Input()` du composant

- `{{ name }}` indique de récupérer la valeur de la propriété name du composant app.

Dans hello.component.ts, le template html affiche aussi `{{name}}` mais là il s'agit du `@Input()`

Interception d'une donnée entrante pour la modifier avant de l'utiliser

On peut utiliser en @Input() une propriété calculée (un getter et un setter) pour modifier la valeur entrante d'une propriété.

Par exemple

```
@Input()
  get name(): string { return this._name; }
  set name(name: string) {
    this._name = (name && name.trim()) || '<no name set>';
  }
  private _name = '';
```

permet d'effacer les espaces en début et fin de chaîne du `name` passé en paramètre et même de lui donner la valeur `'<no name set>'` si aucune valeur n'est passée

Il est également possible de réaliser ceci, ou même des interceptions plus complexes grâce à `ngOnChanges(changes: SimpleChanges)` au lieu d'utiliser des getter et setter.

Parent à l'écoute d'un évènement d'un enfant : `@Ouput` et `EventEmitter`

Principe :

- la vue enfant expose une propriété EventEmitter grâce à `@Output()` un EventEmitter
- quand nécessaire, elle émet l'évènement grâce à `emit()`
- le parent, dans le selector du composant enfant, indique quelle méthode appeler à la réception de l'évènement

Composant enfant voter de `selector` `app-voter` :

```
<button (click)="vote(true)"></button>
@Output() voted = new EventEmitter<boolean>();
vote(agreed: boolean) {
    this.voted.emit(agreed);
}
```

Composant parent :

```
<app-voter (voted)="onVoted($event)"></app-voter>
```

interaction d'un parent avec un enfant via une variable locale

Si l'on a besoin d'accéder à une propriété ou même un méthode d'un enfant, on peut dans la vue parent ajouter dans le selector du composant enfant une variable locale permettant de l'identifier

Exemple, un composant timer et on veut pouvoir le déclencher dans la vue parent qui l'affiche :

```
<app-timer #timer></app-timer>
<button (click)="timer.start()">Start</button>
```

Avantages/Inconvénients :

- méthode simple
- limité au template html, impossible d'accéder aux propriétés et méthodes du composant enfant dans le code de la classe de la vue.

Injection de l'enfant dans le parent :

`@ViewChild()`

Angular permet d'injecter des objects dans d'autres objets via le constructeur (à voir plus tard)

Ici ce n'est pas possible, puisque c'est la vue parent qui crée la vue enfant, elle ne peut donc pas recevoir ce composant en paramètre du constructeur

Heureusement la directive `@ViewChild(composant)` permet de résoudre ce problème :

Dans le code de la classe parent, il suffit de rajouter

```
@ViewChild(TimerComponent)
private timer: TimerComponent
start() { this.timer.start() }
```

et de lier le button à la méthode start()

```
<button (click)="start()">Start</button>
```

Les templates

Texte dynamique

On l'a déjà vu dans les exemples, les délimiteurs `{{` et `}}` permettent d'indiquer que le texte est dynamique et correspond à la valeur de la variable entre les délimiter

`Attention :` ici dynamique est à prendre dans son sens react -> un changement de valeur, changera l'affichage automatiquement

Mais en plus, il peut interpoler des résultat, quelques exemple :

```
{{ 1 + 1 }}
{{ getName() }}
{{ festivals.length + festival.name }}
```

Les variables sont liées à un contexte qui est celui du composant, il ne connait pas les propriétés des vues parents ou des composants enfant

Déclarations et évènements

Les déclarations sont des méthodes ou propriétés que vous pouvez utiliser en réponse à un évènement utilisateur.

Exemple :

```
<button (click)="start()"></button>
```

Les déclarations peuvent se référer à toute donnée du contexte :

```
<button (click)="start($event)"></button>
<button *ngFor="let festival of festivals"
  (click)="select(festival)"></button>
<form #festivalForm (ngSubmit)="onSubmit(festivalForm)">...</form>
```

Pipe : formattage des données

On peut utiliser | et une directive pour formater des donner, par exemple

```
<p>L'année du festival est {{ festival.year | date }}</p>
```

Les formats possibles sont :

- date
- currency
- uppercase
- lowercase
- decimal
- percent

Lien vers une propriété html

Utiliser des [ ] autour d'une propriété indique à angular d'interpréter la chaîne comme une déclaration et non pas comme un littéral

Exemples:

```
<img src="imageUrl">  // affiche l'image qui a pour nom imageUrl
<img [src]="imageUrl"> // affiche l'image dont le nom est la valeur de la variable imageUrl
```

Attention à bien faire la différence entre attribut et propriété :

```
<td> [colspan]="{{1+1}}"> // ERREUR colspan est un attribut, pas une propriété
<td> [colSpan]="{{1+1}}"> // OK : colSpan est la propriété correspondante à l'attribut colspan
```

Autre exemple classique :

```
<button [disabled]="isDisabled">button</button>
```

Rappel : les déclarations utilisées pour modifier les propriétés sont réactive !

Lien bi-directionnel

Il est également possible d'afficher une propriété tout en permettant sa modification

```
<input type="text" [(ngModel)] = 'name' />
```

`ngModel` est une directive faisant à la fois référence à la valeur (value) du input et à l'évènement (input)

Ainsi, si on veut faire un tel lien personnalisé, il faut que le composant

- déclare une variable `@Input`

- déclare un `@Output` `EventEmitter` qui a pour nom, le nom de la variable input+Change

Exemple :

```
@Input()  val: number | string;
@Output() valChange = new EventEmitter<number>();
<app-composant [(val)]="valeur"></app-composant>
```

est un raccourci pour

```
<app-composant [val]="valeur" (valChange)="valeur=$event"></app-composant>
```

Les Directives

Directives

Les directives sont des class ou des instructions permettant d'ajouter des fonctionnalités à votre code html

Elles permettent de gérer les formulaires, les lists, les styles, ...

Il y a 3 types de directives :

1. Directives de composants : on les a déjà vues -> ngOnInit(), @Input, ...
2. Directives d'attributs : changent l'apparence, le comportement d'un élément, d'un composant ou d'une autre directives
3. Directives structurelles : changement la disposition du DOM en ajoutant ou enlevant des éléments

Il est possible de créer ses propres directives. On ne verra ici que celles fournies avec Angular.

Directives d'attribut

Il y en a trois :

1. `NgClass` : ajoute ou enlève un ensemble de classes CSS
2. `NgStyle` : ajoute ou enlève un ensemble de styles HTML
3. `NgModel` : ajoute un lien bidirectionnel à un élément d'un formulaire

NgClass : exemple

app/component.ts

```
currentClasses: {};
/* . . . */
  setCurrentClasses() {
    // CSS classes: added/removed per current state of component properties
    this.currentClasses =  {
      saveable: this.canSave,
      modified: !this.isUnchanged,
      special:  this.isSpecial
    };
  }
```

app.component.html

```
<div [ngClass]="currentClasses">This div is initially saveable,
unchanged, and special.</div>
```

NgStyle : exemple

app/component.ts

```
currentStyles: {};
/* . . . */
  setCurrentStyles() {
    // CSS styles: set per current state of component properties
    this.currentStyles = {
      'font-style':  this.canSave      ? 'italic' : 'normal',
      'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
      'font-size':   this.isSpecial    ? '24px'   : '12px'
    };
  }
```

app.component.html

```
<div [ngStyle]="currentStyles">
  This div is initially italic, normal weight, and extra large
(24px).
</div>
```

NgModel : exemple

app/component.ts

```
import { FormsModule } from '@angular/forms'; // <--- JavaScript
import from Angular
/* . . . */
@NgModule({
/* . . . */

  imports: [
    BrowserModule,
    FormsModule // <--- import into the NgModule
  ],
/* . . . */
})
export class AppModule { }
```

app.component.html

```
<label for="example-ngModel">[(ngModel)]:</label>
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

Directives structurelles

Ces directives rajoutent des fonctionnalités à l'html permettant de manipuler le DOM en en fonction des valeurs des propriétés de la class du composant.

Il y en a trois également:

1. `NgIf` : le classique if-then-else
2. `NgFor` : permet de répéter un nœud du DOM
3. `NgSwitch` : le classique switch-case

NgIf : exemples

NgIf permet d'ajouter ou enlever un élément html en fonction d'une condition

Test avec une valeur booléenne

```
<app-item-detail *ngIf="isActive" [item]="item"></app-item-detail>
```

Test à null

```
<div *ngIf="currentCustomer">Hello, {{currentCustomer.name}}</div>
```

If-then-else

```
<div *ngIf="condition; else elseBlock">Content to render when
condition is true.</div>
<ng-template #elseBlock>Content to render when condition is
false.</ng-template>
```

NgFor : exemple

NgFor permet d'afficher une liste de valeurs très simplement, il suffit de :

1. Lister vos valeurs grâce à *ngFor en affectant les valeurs à une variable locale
2. Définir un block HTML qui dictera comment rendre cette variable

De simples répétitions

```
<div *ngFor="let item of items">{{item.name}}</div>
```

Récupérer l'index de l'itération

```
<div *ngFor="let item of items; let i=index">{{i + 1}} - {{item.name}}</div>
```

N'afficher que certains éléments d'après une condition

```
trackByItems(index: number, item: Item): number { return item.id; }

<div *ngFor="let item of items; trackBy: trackByItems">
  ({{item.id}}) {{item.name}}
</div>
```

NgContainer

Permet de créer un nœud du DOM sans utiliser d'éléments html et donc de ne pas interférer avec le style ou la disposition

Exemple simple

```
<p>
  I turned the corner
  <ng-container *ngIf="hero"> and saw {{hero.name}}. I waved </ng-container>
  and continued on my way.
</p>
```

Exemple plus complexe :

```
<div>
  Pick your favorite hero
  (<label><input type="checkbox" checked (change)="showSad = !showSad">show sad</
label>)
</div>
<select [(ngModel)]="hero">
  <ng-container *ngFor="let h of heroes">
    <ng-container *ngIf="showSad || h.emotion !== 'sad'">
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
    </ng-container>
  </ng-container>
</select>
```

NgSwitch

currentItem est une propriété définie dans le composant

item est une propriété @Input du composant enfant : app-stout

```
<div [ngSwitch]="currentItem.feature">
  <app-stout-item *ngSwitchCase="'stout'" [item]="currentItem"></
app-stout-item>
  <app-device-item *ngSwitchCase="'slim'" [item]="currentItem"></
app-device-item>
  <app-lost-item *ngSwitchCase="'vintage'" [item]="currentItem"></
app-lost-item>
  <div *ngSwitchCase="'bright'"> Are you as bright as
{{currentItem.name}}?</div>
    <!-- . . . -->
  <app-unknown-item *ngSwitchDefault [item]="currentItem"></app-
unknown-item>
</div>
```

Exercice 1 :

Reprenez votre projet Stackblitz et :

1. Initialisez une propriété liste de festivals à la création du composant

```
[
  { name: "FJM2020", tables: 160 },
  { name: "FJM2018", tables: 80 },
  { name: "FJM2019", tables: 110 }
];
```

2. Affichez le nombre de festivals de la liste
3. Affichez la liste des festivals
4. Ajoutez un bouton add qui rajoute un festival prédéterminé
5. Ajoutez un bouton modifier qui modifie le nom du premier festival en lui ajoutant le nombre de tentatives de modification