

Organisation d'une application

- `src/`
 - `app/` -- Composants et services nécessaires au fonctionnement de l'application
 - `assets/` -- Contient les images, fonts, ressources de l'application
 - `config/` -- Fichier(s) de configuration qui seront utilisés par la shared lib de configuration
 - `environments/`
 - `features/` -- Contient les features modules
 - `generated/` -- Contient tous les fichiers générés par des outils de génération de code
 - `shared/` -- Contient tout ce qui est partagé à travers l'application
 - `styles/`

app/

services/

components/

pipes/

.../

app.module.ts

app-routing.module.ts

- Point d'entrée de l'application
- Contient les éléments nécessaires au démarrage de l'application

App module

- Importe d'autres features modules
- Importe le shared module ou des éléments standalone si besoin
- N'exporte rien

features/

```
feature/  
  services/  
  components/  
  .../  
  feature.module.ts  
  feature.routing.module.ts  
other-feature/  
  services/  
  components/  
  .../  
  other-feature.module.ts  
  other-feature.routing.module.ts
```

- Chaque feature représente une partie de l'application

Feature module


- Le feature module est composé de tous les éléments spécifiques à une feature
- Le feature module est importé une seule fois, par le AppModule ou un autre feature module
- Importe le shared module ou des éléments standalone si besoin
- Importe d'autres features modules

shared/

```
components/  
enums/  
types/  
services/  
utils/  
...  
(shared.module.ts)
```

- Contient les éléments réutilisables de l'application
- Les composants réutilisés sont de bon candidats pour être standalone, ce qui évite de faire un shared.module

Shared module

- Exporte tous les éléments réutilisables de l'application
-  Ne pas fournir de providers dans le shared module

environments/

environment.ts

environment.development.ts

- Contient les fichiers spécifiques à un environnement (généralement des constantes)
- Il est possible de générer les fichiers et la configuration associée avec la commande

```
ng generate environments
```

- Exemple de fichier environment.ts

```
export const environment = {  
  production: true,  
};
```

- Et le fichier environment.development.ts associé

```
export const environment = {  
  production: false,  
};
```


environments/

- Pour l'utiliser, on importe le fichier original dans le reste du programme

```
import { environment } from '../environments/environment';
```

```
console.log(environment.production);
```

- Le fichier sera remplacé selon l'environnement

View queries

@ViewChild

- Retour sur les variables de template :

```
<audio #audio src=" ../assets/sample.mp3" ></audio>  
<button (click)="audio.play()">Play</button>  
<button (click)="audio.pause()">Pause</button>
```

- Dans certains cas, on veut pouvoir manipuler les éléments du template dans le composant
- Pour cela, Angular met à disposition les décorateurs d'attributs @ViewChild et @ViewChildren

```
@ViewChild('audio')  
audioRef: ElementRef<HTMLAudioElement>
```

- ViewChild fait une requête sur le template, et assigne le premier résultat correspondant à l'attribut du template

- Il est possible de requêter :
- Un élément du dom avec une variable de template associée

```
@ViewChild('audio')  
audioRef: ElementRef<HTMLAudioElement>
```

- Un composant ou une directive


```
@ViewChild(ChildComponent)  
componentRef: ChildComponent
```

```
@ViewChild(HighlightDirective)  
directiveRef: HighlightDirective
```

- Un provider dans les composants ou directives fils

```
@ViewChild(CounterService)  
childServiceRef: CounterService
```

Quelques précisions sur ViewChild :

- ViewChild renvoie uniquement le premier résultat correspondant à la requête
- Sur le type du retour :
 -  Comme souvent avec les décorateurs, pas de type-checking
 - Pour un composant ou une directive, le type de retour est la classe elle-même
 - Pour un élément HTML, le type de retour est l'interface correspondant encapsulée dans un `ElementRef<T>`
 - Si la requête n'a pas de résultat, le retour est undefined
- La requête n'est pas récursive : elle est limitée au template de composant
- Si au cours de la vie de votre composant le résultat de la requête change, la valeur de l'attribut change également

@ViewChildren

- Le fonctionnement de @ViewChildren est très similaire a @ViewChild, sauf qu'il renvoie tous les résultats de la requête dans une QueryList
- Il est possible de fournir plusieurs sélecteurs, séparés par une virgule
- Comme pour ViewChild, le contenu de la QueryList change avec le composant
- Il est possible de récupérer un observable sur une QueryList qui notifie les subscribers des changements


read

- La propriété `read` permet de changer le type de retour de la requête,
- Exemple pour avoir l'élément du DOM correspondant à un composant fils, plutôt que le composant lui-même

```
@ViewChild(ChildComponent, {read: ElementRef})  
childRef!: ElementRef<HTMLElement>
```

- Ou une directive sur un composant en particulier

```
@ViewChild(ChildComponent, {read: HighlightDirective})  
childDirectiveRef?: HighlightDirective
```

-  Manipuler directement le DOM peut rendre votre application vulnérable

Exercice

- Coder un composant qui permet de choisir une piste audio parmi une liste
- Lorsque l'on change de piste, la lecture actuelle est remise à zero

static (ViewChild seulement)

- Le paramètre static dans le décorateur permet de changer le comportement de la view query
- La requête n'est effectuée qu'une seule fois, après l'initialisation du composant

Lifecycle Hooks

Cycle de vie des composants

- Considérons le composant suivant, qui prends une taille en @Input, et affiche un tableau de cette taille rempli de nombre aléatoires

```
export class TableauComponent {  
  
  @Input({required: true})  
  size!: number  
  
  numberArray: number[] = [...Array(this.size)].map(() => Math.round(Math.random()*10))  
  
}
```

- Un seul élément est crée dans le tableau, pourquoi ?

- Lors de l'appel au constructeur du composant, les bindings ne sont pas encore définis, `this.size` est donc `undefined`

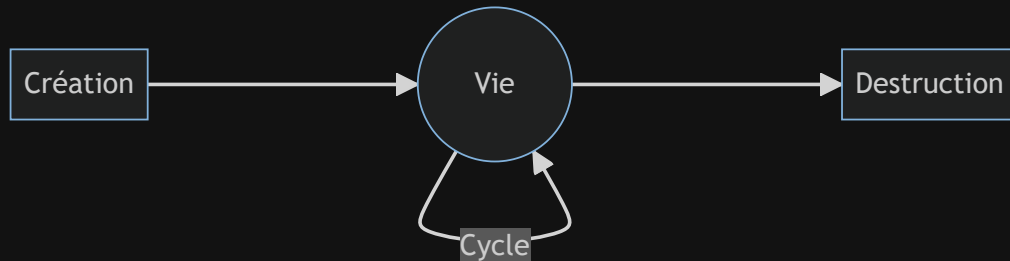
- Autre exemple : avec une View Query

```
@ViewChild('audio')  
audioRef!: ElementRef<HTMLAudioElement>
```

- Dans cet exemple, `audioRef` est encore `undefined` dans le constructeur

```
constructor() {  
  this.audioRef.nativeElement.play()  
}
```

Cycle de vie des composants



- Le cycle de vie d'un composant Angular comporte trois parties :
 - Création du composant
 - Vie du composant (detection cycle)
 - Destruction du composant
- Pour chaque composant, il est possible de définir des hooks, des méthodes qui seront appelées automatiquement à des moments précis du cycle de vie
- Il est également possible de définir des hooks pour des directives, pipes et services

Liste des hooks

Création

- `constructor()`
- `ngOnInit()`
- `ngAfterContentInit()`
- `ngAfterViewInit()`

Vie

- `ngOnChanges()`
- `ngDoCheck()`
- `ngAfterContentChecked()`
- `ngAfterViewChecked()`

Destruction

- `ngOnDestroy()`

- Les hooks de vie du composant sont également appelés lors de la création
- A chaque hook est associé une interface

Hooks de création

ngOnInit()

- Appelé après que les bindings du composants aient été initialisés (et après le premier ngOnChanges)

ngAfterViewInit()

- Appelé une fois après que le template ait été initialisé

Detection de changement

- La vie d'une application Angular est rythmée par les cycles de détection de changement
- A chaque cycle de détection de changement, Angular met à jour l'affichage de l'application
- Un cycle de détection de changements se déclenche lors des évènements suivants :
 - Evènements du DOM surveillé par Angular
 - `setTimeout()` and `setInterval()`
 - Requêtes HTTP

ngOnChanges(changes: SimpleChanges)

- Appelé une première fois après que les attributs @Input soient initialisés
- Puis à chaque fois qu'un attribut @Input est modifié
- Un paramètre SimpleChanges permet de récupérer les changements

```
export declare interface SimpleChanges {  
  [propName: string]: SimpleChange;  
}
```

```
export declare class SimpleChange {  
  constructor(previousValue: any, currentValue: any, firstChange: boolean);  
  
  previousValue: any;  
  currentValue: any;  
  firstChange: boolean;  
  isFirstChange(): boolean;  
}
```

ngDoCheck()

- Appelé à chaque detection cycle
- Peut être utilisé pour détecter des changements que ngOnChanges ne prend pas en compte, comme des mutations d'objets
- Appelé avant qu'Angular ne mette à jour le template
- ⚠ Eviter d'appeler des méthodes coûteuses

ngAfterViewChecked()

- Appelé à chaque detection cycle, une fois que le template ait été mis à jour
- ⚠ Ne pas modifier la valeur des attributs interpolés dans les hooks ngAfterViewInit() et ngAfterViewChecked(), au risque d'avoir un état incohérent entre la valeur des attributs et ce qui est affiché à l'écran

Cycle de détection de changement

- Un évènement déclenche le cycle de détection de changement
- Parent `ngDoCheck()`
- Appel des méthodes du template parent
- Parent view update
- Child `ngDoCheck()`
- Child view update
- Child `afterViewChecked()`
- Parent `afterViewChecked()`
- Appel des méthodes du template parent (uniquement en dev)