Cycle de vie des composants

Temps de lecture : 2 minutes

Le cycle de vie des composants

Les composants suivent un cycle de vie bien défini, composé de plusieurs étapes clés, depuis leur création jusqu'à leur destruction.

`Angular` offre des `hooks` **de cycle de vie** qui permettent aux développeurs d'intervenir à des moments précis de ce cycle pour exécuter du code spécifique.

Voici les grandes étapes du cycle de vie :

1. **Initialisation** : création du composant et initialisation de ses propriétés.
2. **Détection des changements** : vérification et mise à jour des données liées au composant.
3. **Rendu** : affichage du composant dans le DOM.
4. **Destruction** : nettoyage avant la suppression du composant.

Les hooks du cycle de vie

Les `hooks` permettent, par exemple :

- D'effectuer des initialisations lors de la création du composant.
- De réagir aux changements des entrées (`inputs`).
- De nettoyer les ressources lorsque le composant est détruit.

Voici un tableau résumant les phases et méthodes du cycle de vie des composants Angular :

Phase	Méthode	Résumé
Création	`constructor`	Constructeur standard de classe JavaScript. Exécuté lors de l'instanciation du composant par Angular.
Détection des changements	`ngOnInit`	Exécuté une fois après qu'Angular a initialisé toutes les entrées du composant.
	`ngOnChanges`	Exécuté à chaque fois que les entrées du composant changent.
	`ngDoCheck`	Exécuté à chaque vérification des changements dans ce composant.

Phase	Méthode	Résumé
	`ngAfterContentInit`	Exécuté une fois après l'initialisation du contenu projeté dans le composant.
	`ngAfterContentChecked`	Exécuté à chaque vérification des changements dans le contenu projeté.
	`ngAfterViewInit`	Exécuté une fois après l'initialisation de la vue du composant.
	`ngAfterViewChecked`	Exécuté à chaque vérification des changements dans la vue du composant.
Rendu	`afterNextRender`	Exécuté une fois lorsque tous les composants ont été rendus dans le DOM pour la prochaine fois.
	`afterRender`	Exécuté chaque fois que tous les composants ont été rendus dans le DOM.
Destruction	`ngOnDestroy`	Exécuté une fois avant la destruction du composant.

Prenons quelques exemples basiques.

Création : `constructor`

Le constructeur est utilisé pour injecter les dépendances nécessaires.

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

@Component({
  selector: 'app-mon-composant',
  template: `<p>Composant créé !</p>`,
})
export class MonComposant {
  constructor() {
    console.log('Le composant a été créé.');
  }
}
```

Initialisation : `ngOnInit`

Utilisé pour initialiser des propriétés ou effectuer des requêtes `HTTP` après l'initialisation des entrées.

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

```
@Component({
  selector: 'app-produit',
  template: `<p>Produit : {{ produit }}</p>`,
})
export class ProduitComponent implements OnInit {
  produit: string = '';

  ngOnInit() {
    this.produit = 'Ordinateur portable';
    console.log('Produit initialisé.');
  }
}
```

Exemple : gestion d'une liste dynamique avec des composants enfants

Dans cet exemple, un composant parent gère une liste d'éléments affichés à l'aide d'un composant enfant. Nous utilisons les hooks de cycle de vie pour :

1. Charger des données initiales (`ngOnInit`).
2. Réagir aux modifications des données (`ngOnChanges`).
3. Nettoyer les ressources à la destruction (`ngOnDestroy`).

Chaque élément de la liste est représenté par un composant enfant `ListeItemComponent` :

```
import { Component, Input, OnDestroy, OnInit } from '@angular/co
re';

@Component({
  selector: 'app-liste-item',
  template: `
    <li>
      {{ item }}
    </li>
  `,
})
export class ListeItemComponent implements OnInit, OnDestroy {
  @Input() item: string = '';

  ngOnInit() {
    console.log(`Initialisation de l'élément : ${this.item}`);
  }
```

```
  ngOnDestroy() {
    console.log(`Destruction de l'élément : ${this.item}`);
  }
}
```

Le composant parent `ListeComponent` gère la liste des éléments et utilise les hooks pour les charger dynamiquement :

```
import { Component, OnChanges, OnInit, SimpleChanges } from '@an
gular/core';

@Component({
  selector: 'app-liste',
  template: `
    <h2>Liste d'éléments</h2>
    <ul>
      <app-liste-item *ngFor="let element of elements" [item]="e
lement"></app-liste-item>
    </ul>
    <button (click)="ajouterElement()">Ajouter un élément</butto
n>
    <button (click)="supprimerElement()">Supprimer le dernier él
ément</button>
  `,
})
export class ListeComponent implements OnInit, OnChanges {
  elements: string[] = [];

  ngOnInit() {
    console.log('Chargement initial de la liste');
    this.elements = ['Élément 1', 'Élément 2', 'Élément 3'];
  }

  ngOnChanges(changes: SimpleChanges) {
    if (changes['elements']) {
      console.log('La liste a été modifiée.');
    }
  }

  ajouterElement() {
    const nouvelElement = `Élément ${this.elements.length + 1}`;
    this.elements.push(nouvelElement);
    console.log(`Ajout : ${nouvelElement}`);
  }
```

```
  supprimerElement() {
    const dernierElement = this.elements.pop();
    console.log(`Suppression : ${dernierElement}`);
  }
}
```

Initialisation (`ngOnInit`) :

- Le composant parent charge une liste initiale d'éléments.
- Le composant enfant utilise `ngOnInit` pour loguer chaque élément lorsqu'il est créé.

Ajout dynamique : lorsque l'utilisateur clique sur "Ajouter un élément", un nouvel élément est ajouté à la liste, et un nouveau composant enfant est créé.

Suppression dynamique :

- Lorsque l'utilisateur clique sur "Supprimer le dernier élément", le dernier composant enfant est détruit.
- Le composant enfant utilise `ngOnDestroy` pour loguer sa destruction.

Modification détectée (`ngOnChanges`) : Si des modifications sont apportées à la liste des éléments, le `hook ngOnChanges` peut être utilisé pour réagir.

Exemple de la vidéo

`DymaComponent`

```
import { Component, SimpleChanges, afterNextRender, afterRender
} from '@angular/core';

@Component({
  selector: 'app-dyma',
  template: `
    <p>Composant Dyma</p>
  `,
})
export class DymaComponent {
  ngOnChanges(changes: SimpleChanges) {
    console.log('ngOnChanges');
    console.log(changes);
  }

  constructor() {
```

```
    console.log('constructor');

    afterNextRender(() => {
      console.log('after next render');
    });
    afterRender(() => {
      console.log('after render');
    });
  }

  ngOnInit() {
    console.log('ngOnInit');
  }

  ngAfterContentInit() {
    console.log('ngAfterContentInit');
  }

  ngAfterContentChecked() {
    console.log('ngAfterContentChecked');
  }

  ngAfterViewInit() {
    console.log('ngAfterViewInit');
  }

  ngAfterViewChecked() {
    console.log('ngAfterViewChecked');
  }

  ngOnDestroy() {
    console.log('ngOnDestroy');
  }
}
```

AppComponent

HTML :

```
@if (test()) {
  <app-dyma></app-dyma>
}
```

TypeScript :

```ts
import { Component, signal } from '@angular/core';
import { DymaComponent } from './components/dyma.component';

@Component({
  selector: 'app-root',
  imports: [DymaComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  test = signal(true);

  constructor() {
    setTimeout(() => {
      this.test.set(false);
    }, 3000);
  }
}
```

`DymaComponent` **s'affiche** lorsque `test()` est vrai.

Après 3 secondes, `test` est défini sur `false`, ce qui provoque la destruction de `DymaComponent`.

Les logs des `hooks` de cycle de vie apparaissent dans la console :

- lors de l'initialisation de `DymaComponent` : `constructor`, `ngOnInit`, `ngAfterContentInit`, `ngAfterContentChecked`, `ngAfterViewInit`, `ngAfterViewChecked`, et les hooks de rendu (`after next render`, `after render`).
- lors de la destruction de `DymaComponent` : `ngOnDestroy`.