Cycle de détection de changement

- Un évènement déclenche le cycle de détection de changement

- Parent ngOnChanges()
- Parent ngDoCheck()
- Appel des méthodes du template parent
- Parent view update

- Child ngOnChanges()
- Child ngDoCheck()
- Appel des méthodes du template child
- Child view update

- Child afterViewChecked()
- Parent afterViewChecked()

Hooks de destruction

ngOnDestroy()

- Appelé une fois lors de la destruction du composant
- Peut être utilisé pour unsubscribe à des Observables

ExpressionChangedAfterItHasBeenCheckedError

- Les méthodes dans le tempalte sont appelées deux fois en développement
- Angular effectue un controle pour s’assurer que rien n’ait changé après la mise à jour de l’affichage
- L’erreur NG0100: ExpressionChangedAfterItHasBeenCheckedError, est lancée par Angular en développement uniquement si une valeur change après que la détéction ait eu lieu

ChangeDetectionStrategy

- Lors d’un cycle de détection de changement, Angular ne sais pas quels composants doivent être actualisés
- Angular à choisi par défaut de faire une vérification sur tous les composants à chaque cycle
- Cette méthode à l’avantage d’être simple au développeur à implémenter
- Par contre elle peut conduire à des problèmes de performances, surtout pour des grosses applications

ChangeDetectionStrategy.OnPush

- La stratégie OnPush permet de réduire le nombre de vérification à faire lors de la détection de changement
- L'option se configure au niveau des métadonnées du décorateur @Component

```
changeDetection: ChangeDetectionStrategy.OnPush
```

- Le composant possède un booléen *dirty*, qui symbolise la nécessité d'actualiser le composant au prochain cycle de détection de changement
- Les hooks ngDoCheck() et ngAfterViewChecked() d'un composant sont appelés, même si celui-ci n'est pas dirty

ChangeDetectionStrategy.OnPush

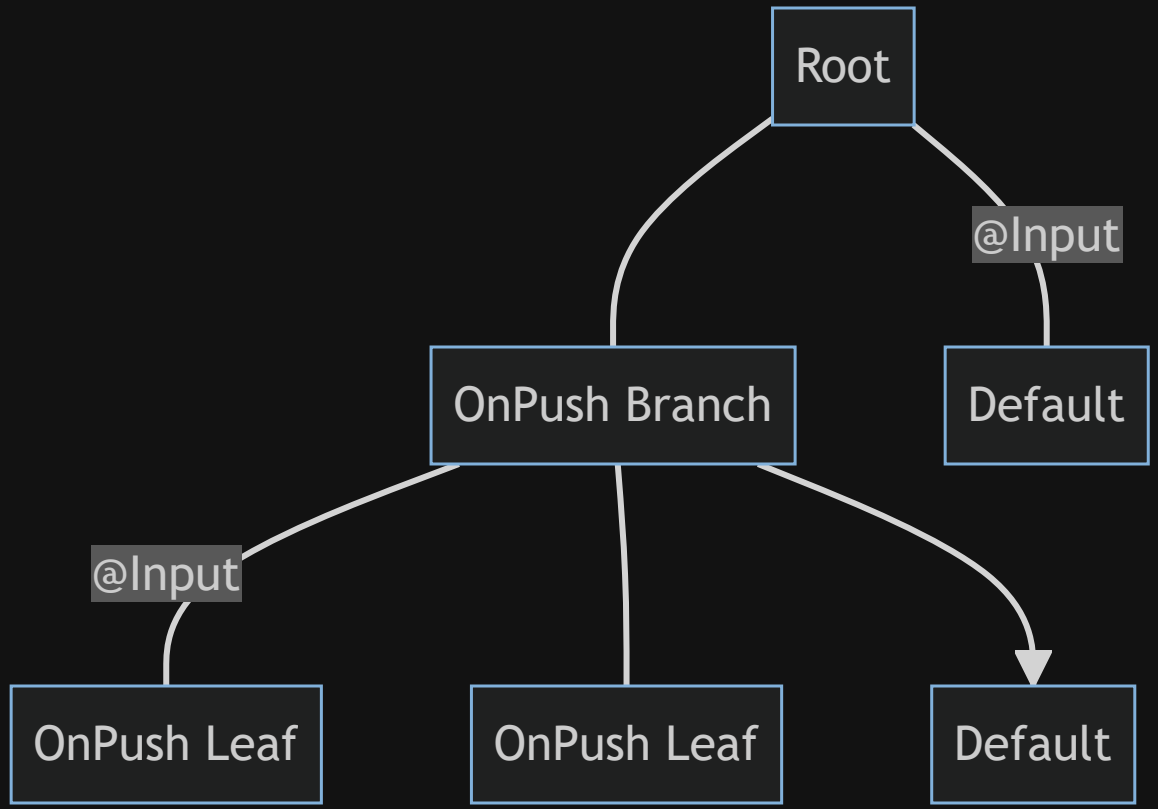
- Un composant est marqué automatiquement à dirty avec les actions suivantes
  - Un évènement du DOM lié à ce composant à lieu
  - Une variable @Input ou @Output de ce composant change
- Lorsque un composant est marqué dirty, tous ses parents sont égalements marqués
- Il est également possible de marquer un composant dirty manuellement en injectant le service ChangeDetectorRef

```
cdr = inject(ChangeDetectorRef)
```

- Et en utilisant la méthode markForCheck()

```
this.cdr.markForCheck()
```

Exemple


Root
@Input
OnPush Branch
Default
@Input
OnPush Leaf
OnPush Leaf
Default

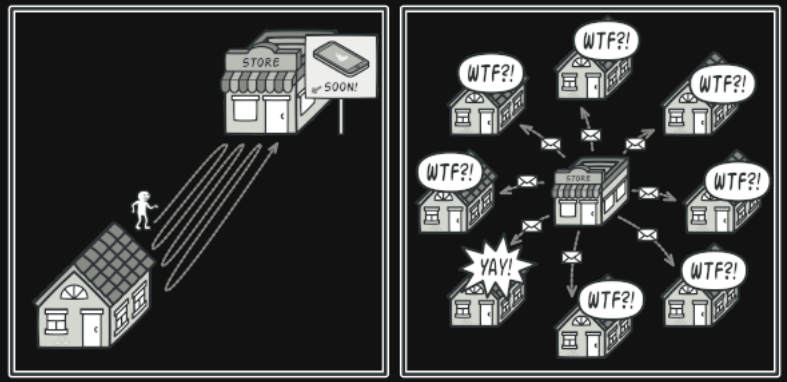
Limites

- Certaines actions courrantes dans une application Angular ne mettent pas automatiquement le composant à dirty
  - Mutations d’objets dans un @Input
  - setTimer et setInterval
  - Services

RxJS et Observables

Design pattern Observer

- Mise en situation : deux objets, un Customer et un Store
- Le Customer est intéressé par un objet, qui devrait être disponible *bientôt*



- Le Customer peut rendre visite regulièrement au Store pour savoir si le produit est disponible
  - Beaucoup de temps perdu
- Le Store peut peut envoyer à tous les Customer une notification lorsque un produit est disponible
  - Beaucoup de messages inutiles

Design pattern Observer

- Le design pattern Observer propose une solution à ce problème :
  - Notre Customer s’abonne au Store
  - Le Store maintient une liste des Customer abonnés
  - Lorsque le Store a un nouveau produit, il notifie les Customer aboonés

- https://refactoring.guru/design-patterns/observer

RxJS et Observable

- En se basant sur çe concept, la librairie RxJS permet de gérér les séquences d'évnènements asynchrones
- L'élément principal est l'Observable qui peut :
  - Émettre de 0 à une infinité de valeurs
  - Peut être en erreur
  - Peut être complété
- Un observable en état d'erreur ne va plus jamais émettre de valeur, ni passer en état de complétion.
- Un observable qui a complété ne va plus jamais émettre de valeur, ni passer en état d'erreur.

Créer un Observable

- La classe (générique) RxJS Observable représente un observable
- Il est possible de créer un observable en utilisant le constructeur

```
const observable = new Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});
```

- next() pour envoyer une valeur (synchrone ou asynchrone)
- error() pour envoyer une erreur
- complete() pour signaler que l'observable est terminé

Créer un Observer

- L'interface (également générique) RxJS Observer représente un observateur qui va s'abonner à un observable et consommer ses valeurs

```
const observer: Observer<number> = {
  next: x ⇒ console.log('Observer got a next value: ' + x),
  error: err ⇒ console.error('Observer got an error: ' + err),
  complete: () ⇒ console.log('Observer got a complete notification'),
};
```

- Les callbacks next, error et complete sont appelés lors des émissions respectives de l'observable
- Il est possible de définir un observateur partiel

Souscrire à un observable

- On peut s'abonner à un observable en utilisant la méthode subscribe() de celui çi

```
subscription: Subscription = this.observable$.subscribe(this.observer)
```

- La classe RxJS Subscription représente un abonnement, retour de la métode subscribe()
- Un observable est dit *lazy*, c'est à dire qu'il n'est exécuté que lors de l'abonnement d'un observateur
- L'observable s'execute pour chaque abonnement
- Un observable peut émettre potentiellement à l'infini, on peut utiliser l'abonnement pour se désabonner

```
this.subscription.unsubscribe()
```

- Dans mon exemple précédent, que ce passe t'il si j'ajoute un console.log() dans le setInterval ?

```
setInterval(() ⇒ {
    console.log("tick")
    subscriber.next(4);
}, 1000);
```

- Malgré que l'abonnement se soit arrété, le log continue de s'affichier
- Compléter l'observable ne résout pas le problème non plus

TeardownLogic

- Il est nécessaire de libérer manuellement certaines ressources utilisées par les observables
- Un exemple : setInterval()
- Pour cela, on peut définir une TeardownLogic qui libère les ressources à de la fin de l'observable

```
const observable = new Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  let intervalId = setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
  return () => clearInterval(intervalID)
});
```

Async pipe

- Angular met à disposition le pipe async pour gérer facilement les observables
- Le pipe async s'abonne à un Observable (ou une Promise) et renvoie la dernière valeur émise
- A chaque utilisation du pipe, un nouvel abonnement est fait à l'observable

- Composant :

```
obs$ = interval(10)
```

- Template :

```
{{ obs$ | async }}
```

- Le pipe gère automatiquement le désabonnement à l'observable
- ⚠ à l'utilisation du pipe async avec le *nglf

Exercice

- Nous allons tenter de recréer une version simplifiée du pipe async (uniquement avec des Observables)
- Qu’est ce que fait le pipe async ?
  - On passe un observable au pipe dans le template (pipeTransform)
  - Le pipe async gère l’abonnement et le désabonnement de l’observable
  - Le pipe async retourne la dernière valeur émise par l’observable
- Comment gérer le désabonnement ?
- Est ce que le pipe async est pur ou impur ?
- Plus avancé : que ce passe t’il lorsque je change l’observable dans le template ?