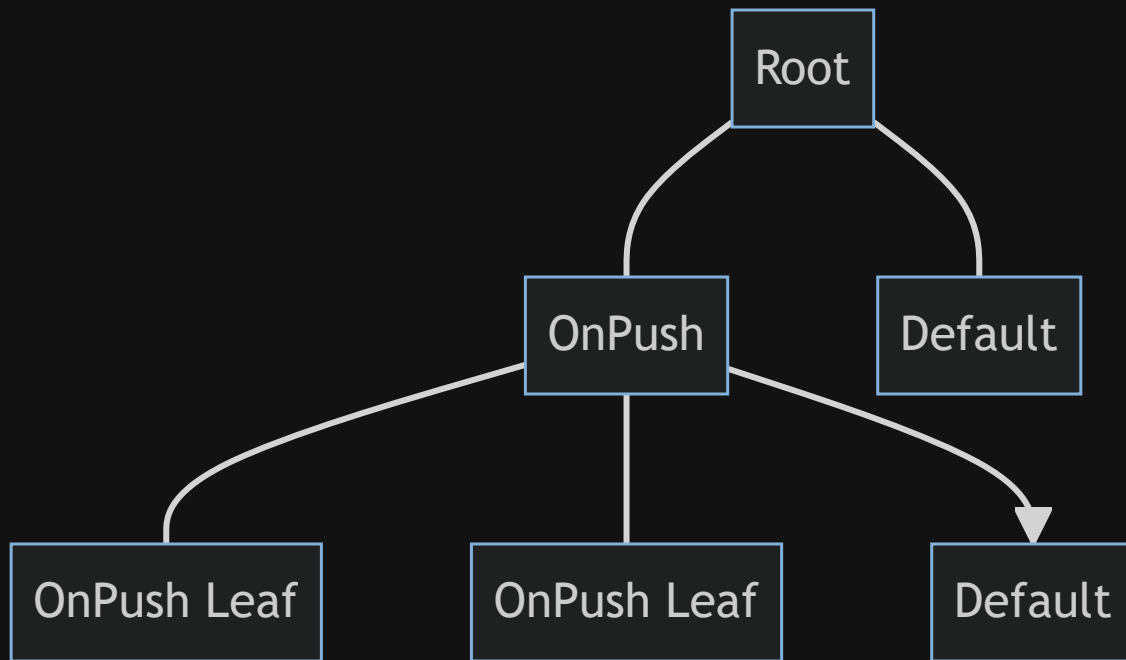


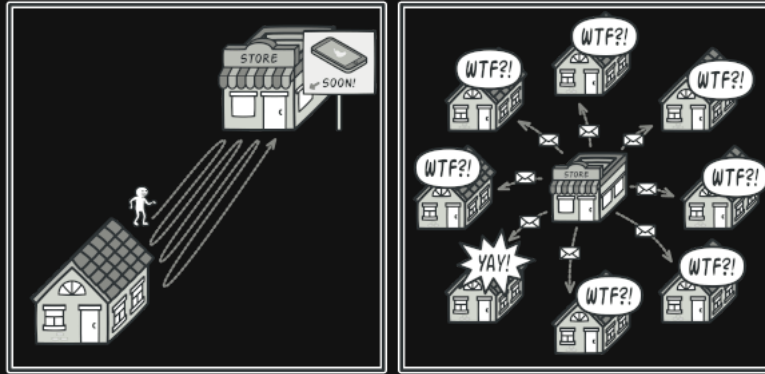
Exemple



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 régulièrement au Store pour savoir si le produit est disponible
 - Beaucoup de temps perdu
- Le Store 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 abonnés
- <https://refactoring.guru/design-patterns/observer>

RxJS et Observable

- En se basant sur ce concept, la librairie RxJS permet de gérer les séquences d'événements 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-ci

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

- La classe RxJS Subscription représente un abonnement, retour de la méthode `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'exécute 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 arrêté, le log continue de s'afficher
- 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 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)  
});
```

"Hot" et "Cold" observable

- On distingue le producteur (celui qui produit les données) de l'observable (responsable de la distribution des données)
- Un observable est dit "cold" lorsque un producteur est créé par l'observable pour chaque abonnement
- Un observable est dit "hot" lorsque le producteur est créé en dehors de l'abonnement
- Dans la plupart des cas, cela implique que l'observable "cold" est unicast, alors que l'observable "hot" est multicast

Opérateurs de création

- En pratique, on crée rarement un observable nous même
- RxJS fournit des opérateurs pour créer des observables
- Angular fournit également des observables (HttpClient, ReactiveForms, ...)

EMPTY, NEVER

```
const EMPTY: Observable<never>;
```

- EMPTY complète immédiatement sans émettre de valeur ni d'erreur

```
const NEVER: Observable<never>;
```

- NEVER n'émet rien, ni ne complète

of(), from()

```
of<T>(...args: T[]): Observable<T>
```

- of() crée un observable à partir d'une liste de valeurs
- from() crée un observable à partir d'un tableau, une promesse, un objet itérable ou "observable-like"
- Quelle est la différence entre les deux appels :

```
of([1,2,3])  
from([1,2,3])
```

timer(), interval()

```
timer(due: number | Date): Observable<0>
```

- Crée un observable qui attend un temps donné (en ms), ou une date, et émet 0

```
interval(period: number = 0): Observable<number>
```

- Crée un observable qui émet une séquence incrémentale à une certaine période

```
timer(startDue: number | Date, intervalDuration: number): Observable<number>
```

- Crée un observable qui attend un temps donné (en ms), ou une date, puis émet une séquence incrémentale à une certaine période

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 *ngIf

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 pure ou impur ?
- Plus avancé : que ce passe t'il lorsque je change l'observable dans le template ?