

reduce(), scan()

```
reduce<V, A>(  
  accumulator: (acc: V | A, value: V, index: number) => A,  
  seed?: any  
) : OperatorFunction<V, V | A>
```

- Applique une fonction d'accumulation sur les valeurs émises par la source
 - La valeur renvoyée par la fonction d'accumulation est assignée au paramètre acc de l'appel suivant
 - Il est possible de définir une seed, valeur initiale de l'accumulateur
- reduce() émet une seule fois, lorsque l'observable source complète, alors que scan() émet la valeur accumulée à chaque émission de la source

tap()

```
tap(observerOrNext?: (value: T) => void): MonoTypeOperatorFunction<T>
```

- L'opérateur tap() permet de créer un effet de bord sur un opérateur
- Les valeurs émises par l'observable créé sont les mêmes que l'observable source
- tap() est très utile pour déboguer vos observables
- ⚠ Si votre opérateur source émet des objets, ils peuvent être mutés dans le tap()

Higher-order Observables

- Retour sur notre exemple d'autocomplétion
- Que ce passe t'il si j'essaie de combiner nos deux observables avec un map ?

```
...
clients$: Observable<Client[]>

ngAfterViewInit() {
  this.clients$ = fromEvent(this.input.nativeElement, 'input').pipe(
    map(ev => this.clientService.getFilteredSortedClients((ev.target as HTMLInputElement).value))
  )
}
...
```

- Il y a une erreur de compilation, c'est normal, car notre map ne renvoie pas un Observable<Client[]>, mais un Observable<Observable<Client[]>>

Higher-order Observables

- Un observable qui envoie des observables est appelé un *higher-order observable*
- Dans la plupart des cas, un tel observable n'est pas utilisable directement, on veut convertir cet observable en un observable émettant des valeurs
- Cette opération (appelée *flatten*) est possible via des opérateurs

mergeAll(), concatAll(), switchAll(), exhaustAll()

- Ces 4 opérateurs s'abonnent à tous les observables générés par l'higher-order observable, et émettent les valeurs émises par les observables intermédiaires
- Il est tellement courant d'utiliser ces opérateurs après un map(), que des opérateurs spécifiques existent, ainsi :

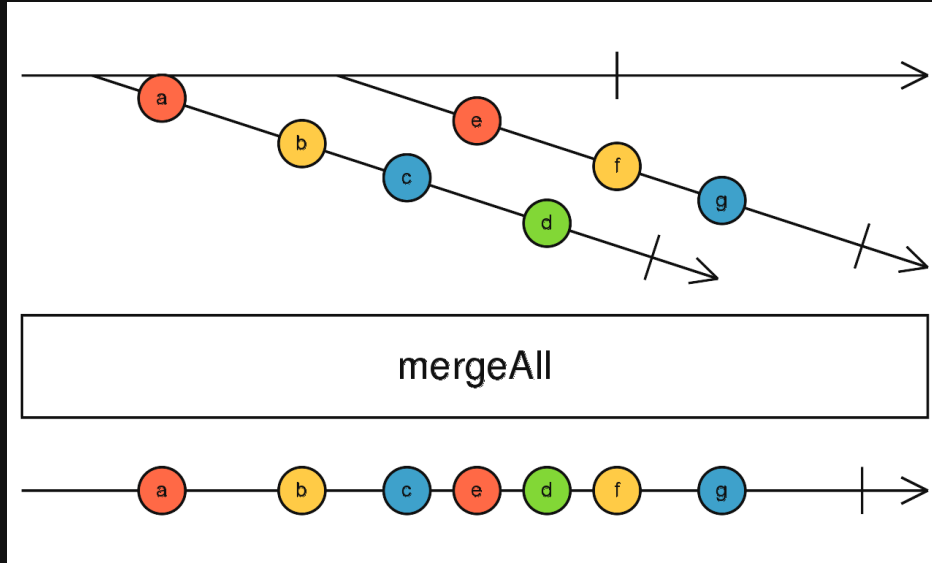
```
obs$.pipe(  
  map(callback),  
  mergeAll()  
)
```

- Peut être simplifié en :

```
obs$.pipe(  
  mergeMap(callback),  
)
```

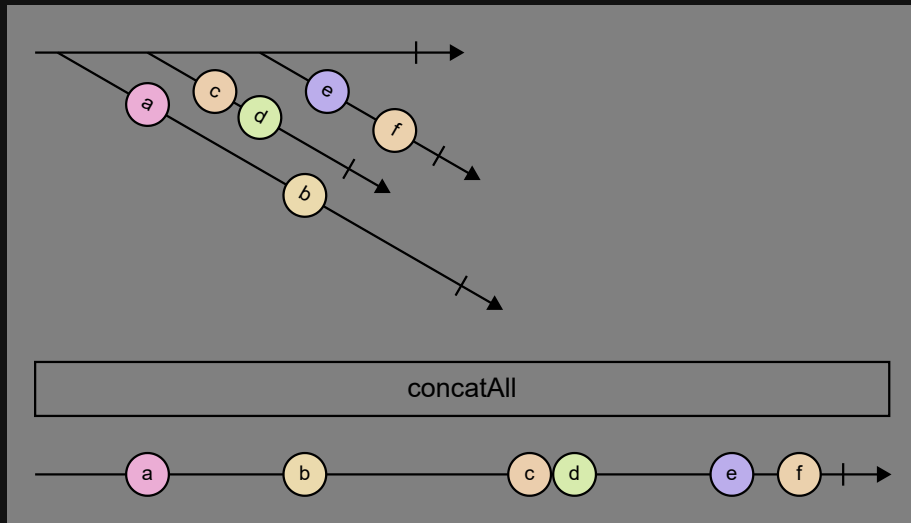
- Il est également possible d'utiliser concatMap(), switchMap(), exhaustMap() de la même manière
- Ces 4 opérateurs sont très similaires, mais ont quelques différences, que nous allons voir maintenant

mergeAll()



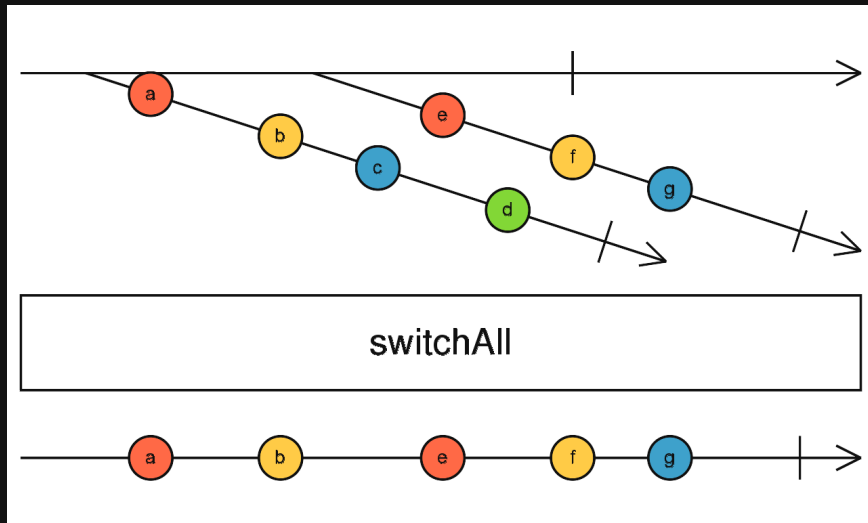
- `mergeAll()` s'abonne à tous les observables et émet toutes les valeurs émises au fur et à mesure

concatAll()



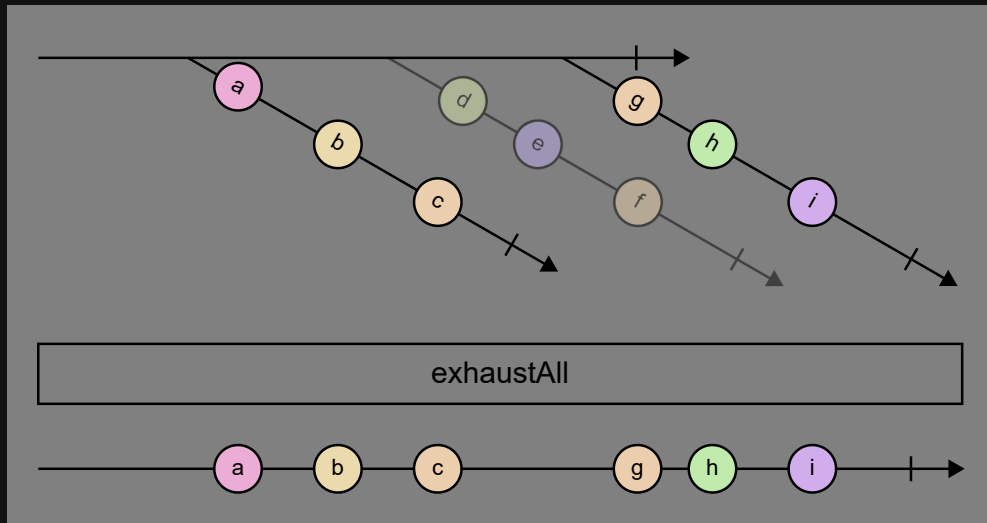
- N'émet les valeurs que d'un observable à la fois, `concatAll()` attend que le premier ait fini d'émettre avant de passer au deuxième, puis au troisième, ...
- Si le premier observable ne complete jamais, les suivants ne seront jamais pris en compte
- Garde en mémoire une file de tous les observables qu'il a à traiter

switchAll()



- N'émet les valeurs que d'un observable à la fois
- À chaque fois qu'un nouvel observable est émis, `switchAll()` se désabonne à l'observable en cours et passe au nouvel observable
- Une partie des valeurs émises par les observables intermédiaires sont perdues

exhaustAll()



- N'émet les valeurs que d'un observable à la fois
- `exhaustAll()` va ignorer tous les observables émis tant que l'observable en cours n'a pas complété
- Une partie des observables intermédiaires sont perdues

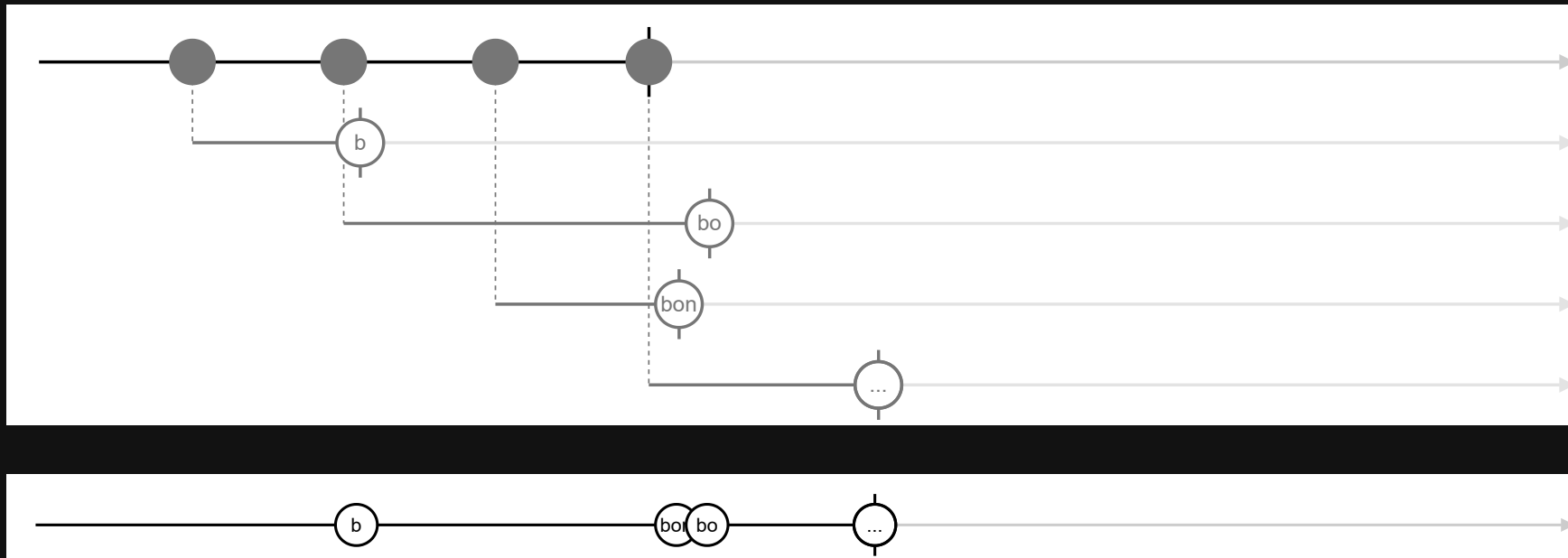
Exemple

- De retour sur notre exemple d'autocomplétion
- Nous allons devoir utiliser un de ces opérateurs pour lier nos deux observables ensemble, mais lequel ?
- Avec notre serveur en local, la réponse sera à chaque fois très rapide. Ce qui n'est pas toujours le cas en pratique
- La différence de comportement entre les opérateurs va devenir cruciale si le serveur à un délai de réponse, que l'on va pouvoir simuler avec json-server

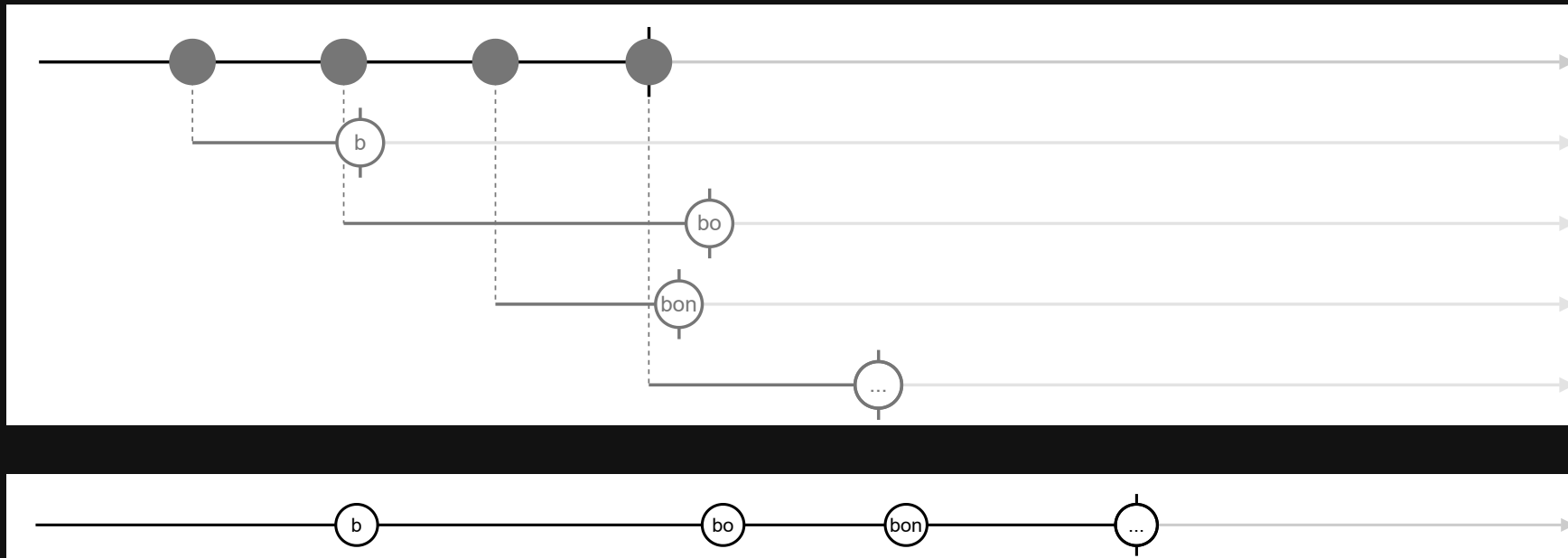
`--delay, -d`

Add delay to responses (ms)

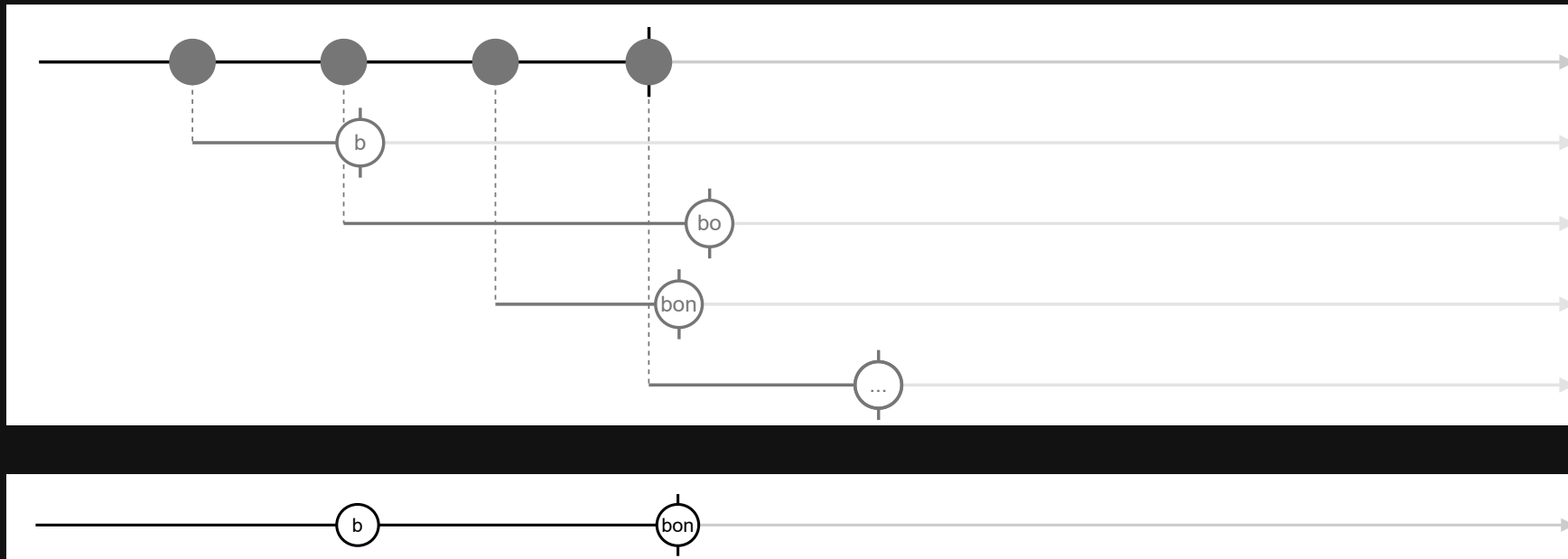
Avec mergeMap()



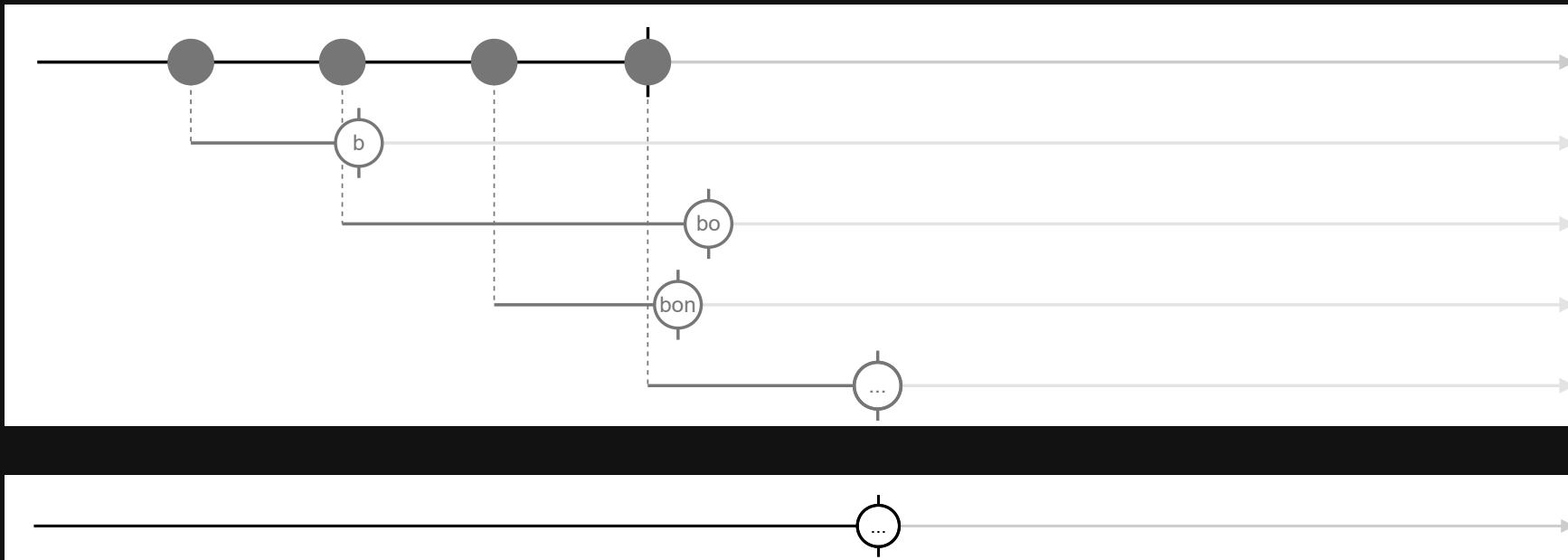
Avec concatMap()



Avec exhaustMap()



Avec switchMap()



Exercice :

- Implémenter l'observable game\$:
 - Lorsque l'on s'abonne à game\$, un timer de 5 secondes est déclenché
 - Pendant ce timer, game\$ compte tous les clicks utilisateur faits sur le document
 - A la fin du timer, game\$ envoie le nombre de clicks, et complète

Implémentation finale

- Pour ce cas, l'opérateur à utiliser est le `switchMap`
- Quelques idées pour améliorer le composant d'autocomplétion :
 - Attendre que l'utilisateur arrête de taper pour lancer la requête
 - Ne pas faire deux fois la même requête de suite
 - Attendre 3 caractères pour lancer la requête