Angular : RxJS

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Angular

Programmation réactive

- paradigme de programmation orienté flux de données et propagation des changements
- inspiré du patron de conception observable
- Deux concepts importants :
  - **Observable** : une fonction retournant un flux de valeurs à un observateur de manière synchrone ou asynchrone. Un observable s'exécute s'il y a un observateur (**observer**) et un abonnement (avec la méthode `subscribe()`)
  - **Observer** : un élément (objet) qui se souscrit (**subscribe()**) à un observable pour recevoir les changements et exécuter une suite de code

Angular

Programmation réactive

- paradigme de programmation orienté flux de données et propagation des changements
- inspiré du patron de conception observable
- Deux concepts importants :
  - **Observable** : une fonction retournant un flux de valeurs à un observateur de manière synchrone ou asynchrone. Un observable s'exécute s'il y a un observateur (**observer**) et un abonnement (avec la méthode `subscribe()`)
  - **Observer** : un élément (objet) qui se souscrit (**subscribe()**) à un observable pour recevoir les changements et exécuter une suite de code

RxJS : Reactive extensions for JavaScript

Angular

La méthode `subscribe()` prend trois paramètre

1. la première se déclenche à chaque fois que l'observable émet de nouvelles données (ces données sont reçues comme paramètre)
2. la deuxième se déclenche si l'observable émet une erreur, et reçoit cette erreur comme paramètre
3. la troisième se déclenche lorsque l'observable se termine, et ne reçoit aucun paramètre.

Angular

Pour commencer

- créez un composant `observable`
- ajoutez `<app-observable></app-observable>` dans `app.component.html`

Le fichier `observable.component.ts`

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

@Component({
  selector: 'app-observable',
  templateUrl: './observable.component.html',
  styleUrls: ['./observable.component.css']
})
export class ObservableComponent implements OnInit {

  status = '';
  tab: Array<number> = [];
  constructor() { }
  ngOnInit() { }
}
```

Le fichier `observable.component.ts`

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

@Component({
  selector: 'app-observable',
  templateUrl: './observable.component.html',
  styleUrls: ['./observable.component.css']
})
export class ObservableComponent implements OnInit {

  status = '';
  tab: Array<number> = [];
  constructor() { }
  ngOnInit() { }
}
```

Le fichier `observable.component.html`

```
<h1>éléments</h1>
<ul>
    <li *ngFor="let elt of tab">
        {{ elt }}
    </li>
</ul>
<div>{{ status }}</div>
```

Angular

Pour créer un observable, on peut utiliser la méthode `of()` qui convertit un ensemble de paramètres en `observable`

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

@Component({
  selector: 'app-observable',
  templateUrl: './observable.component.html',
  styleUrls: ['./observable.component.css']
})
export class ObservableComponent implements OnInit {

  status = '';
  tab: Array<number> = [];
  constructor() { }
  ngOnInit(): void {
    const observable: Observable<number> = of(1, 2, 3);
  }
}
```

Angular

On peut utiliser `from()` pour construire un observable à partir d'un tableau

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

@Component({
  selector: 'app-observable',
  templateUrl: './observable.component.html',
  styleUrls: ['./observable.component.css']
})
export class ObservableComponent implements OnInit {

  status = '';
  tab: Array<number> = [];
  constructor() { }
  ngOnInit() {
    const tableau = [1, 2, 3];
    const observable: Observable<number> = from(tableau);
  }
}
```

Angular

On peut utiliser `range()` pour définir un interval de nombre entier

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

@Component({
  selector: 'app-observable',
  templateUrl: './observable.component.html',
  styleUrls: ['./observable.component.css']
})
export class ObservableComponent implements OnInit {

  status = '';
  tab: Array<number> = [];
  constructor() { }
  ngOnInit() {
    const observable: Observable<number> = range(1, 3);
  }
}
```

Angular

Lorsqu'un observateur s'abonne à notre observable, il peut implémenter trois méthodes pour spécifier ce qu'il faut faire

```
ngOnInit() {
  const tableau = [1, 2, 3];
  const observable: Observable<number> = from(tableau);
  const observer: Observer<number> = {
    next: (value) => {
      this.tab.push(value);
    },
    error: (error) => {
      this.status = error;
    },
    complete:    () => {
      this.status = 'fini';
    }
  };

}
```

Angular

On peut aussi directement faire

```
ngOnInit() {
  const tableau = [1, 2, 3];
  const observable: Observable<number> = from(tableau);
  observable.subscribe(
    (value) => {
      this.tab.push(value);
    },
    (error) => {
      this.status = error;
    },
    () => {
      this.status = 'fini';
    }
  );
}
```

L'émission de valeurs s'effectue d'une manière synchrone et par conséquent on ne voit pas la réception des éléments dans le navigateur.

Angular

Pour avoir une exécution asynchrone, on utilise la fonction `interval(1000)` une infinité de valeurs incrémentielles commençant de 0 : une valeur par seconde

```
ngOnInit() {
  const observable: Observable<number> = interval(1000);
  observable.subscribe(
    (value) => {
      this.tab.push(value);
    },
    (error) => {
      this.status = error;
    },
    () => {
      this.status = 'fini';
    }
  );
}
```

Pour avoir une exécution asynchrone, on utilise la fonction `timer(3000, 1000)` qui envoie une infinité de valeurs incrémentielles commençant de 0 : une valeur par seconde la première valeur sera envoyée après 3 secondes

```
ngOnInit() {
  const observable: Observable<number> = timer(3000, 1000);
  observable.subscribe(
    (value) => {
      this.tab.push(value);
    },
    (error) => {
      this.status = error;
    },
    () => {
      this.status = 'fini';
    }
  );
}
```

Les deux fonctions `timer()` et `interval()` ne se terminent jamais.

Pour éviter les problèmes de mémoire, il faut penser à se désabonner à la destruction du composant

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Observable, Subscription, timer } from 'rxjs';
@Component({
  selector: 'app-observable',
  templateUrl: './observable.component.html',
  styleUrls: ['./observable.component.css']
})
export class ObservableComponent implements OnInit, OnDestroy {
  tab: Array<number> = [];
  status = '';
  constructor() { }
  subscription: Subscription;
  ngOnInit() {
    const observable: Observable<number> = timer(3000, 1000);
    this.subscription = observable.subscribe(
      (value) => { this.tab.push(value); },
      (error) => { this.status = error; },
      () => { this.status = 'fini'; }
    );
  }
  ngOnDestroy() { this.subscription.unsubscribe(); }
}
```

Angular

Pour indiquer le nombre d'élément à émettre, on utilise la méthode `pipe()` qui nous permet de faire appel à l'opérateur `take()`

```
ngOnInit() {
  const observable: Observable<number> = interval(1000).pipe(take(10));

  observable.subscribe(
      (value) => {
        this.tab.push(value);
      },
      (error) => {
        this.status = error;
      },
      () => {
        this.status = 'fini';
      }
  );
}
```

On peut importer les opérateurs ainsi : `import { take } from 'rxjs/operators';`

Angular

On peut combiner les opérateurs en appliquant une modification sur les 10 éléments reçus

```
ngOnInit() {
  const observable: Observable<number> = interval(1000).pipe(
          take(10),
          map(elt => elt + 3)
  );

  observable.subscribe(
       (value) => {
         this.tab.push(value);
       },
       (error) => {
         this.status = error;
       },
       () => {
         this.status = 'fini';
       }
  );
}
```

Angular

On peut aussi filtrer les éléments pairs

```
ngOnInit() {
  const observable: Observable<number> = interval(1000).pipe(
          take(10),
          map(elt => elt + 3),
          filter(elt => elt % 2 === 0)
  );

  observable.subscribe(
       (value) => {
         this.tab.push(value);
       },
       (error) => {
         this.status = error;
       },
       () => {
         this.status = 'fini';
       }
  );
}
```

Angular

On peut aussi compter les éléments selon un critère

```
ngOnInit() {
  const observable: Observable<number> = interval(1000).pipe(
          take(10),
          count(elt => elt % 2 === 0)
  );

  observable.subscribe(
       (value) => {
         this.tab.push(value);
       },
       (error) => {
         this.status = error;
       },
       () => {
         this.status = 'fini';
       }
  );
}
```

Angular

On peut sélectionner le maximum

```
ngOnInit() {
  const observable: Observable<number> = interval(1000).pipe(
          take(10),
          max()
  );

  observable.subscribe(
      (value) => {
        this.tab.push(value);
      },
      (error) => {
        this.status = error;
      },
      () => {
        this.status = 'fini';
      }
  );
}
```

Angular

On peut fusionner plusieurs observables grâce à la fonction `merge()`

```
ngOnInit() {
  const tableau = [1, 2, 3];
  const observable1: Observable<number> = of(1, 2, 3);
  const observable2: Observable<number> = of(4, 5, 6);
  const merged = merge(
      observable1,
      observable2
  );

  merged.subscribe(
      (value) => {
        this.tab.push(value);
      },
      (error) => {
        this.status = error;
      },
      () => {
        this.status = 'fini';
      }
  );
}
```

Angular

Subject

- Il a à la fois le rôle d'un observateur et d'un observable
- Il autorise la souscription de plusieurs observateurs

Angular

Dans `ngOnInit()`, commençons par déclarer un `Subject`

```
const subject = new Subject<number>();
```

Angular

Dans `ngOnInit()`, commençons par déclarer un `Subject`

```
const subject = new Subject<number>();
```

Plusieurs observateurs peuvent s'abonner à notre `subject`

```
subject.subscribe({
    next: (value) => console.log(`A : ${value}`)
});
subject.subscribe({
    next: (value) => console.log(`B : ${value}`)
});
```

Angular

Dans `ngOnInit()`, commençons par déclarer un `Subject`

```
const subject = new Subject<number>();
```

Plusieurs observateurs peuvent s'abonner à notre `subject`

```
subject.subscribe({
    next: (value) => console.log(`A : ${value}`)
});
subject.subscribe({
    next: (value) => console.log(`B : ${value}`)
});
```

Introduisons deux nouvelles valeurs à notre `subject`

```
subject.next(1);
subject.next(2);
// le résultat est
// A : 1
// B : 1
// A : 2
// B : 2
```

Un `Subject` est aussi un observateur, il peut donc s'abonner à un observable

```
ngOnInit() {
    const subject = new Subject<number>();
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });


    observable.subscribe(subject);
}
```

Un `Subject` est aussi un observateur, il peut donc s'abonner à un observable

```
ngOnInit() {
    const subject = new Subject<number>();
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });


    observable.subscribe(subject);
}
```


Le résultat est

```
// A : 1
// B : 1
// A : 2
// B : 2
// A : 3
// B : 3
```

Angular

Remarque

- Un observable ne peut avoir qu'un seul observateur
- Un observable peut utiliser l'opérateur `multicast` et un `subject` pour avoir plusieurs observateurs
- L'observable devient ainsi un `ConnectableObservable` et utilise la méthode `connect` pour propager les changements

Exemple

```
const observable: Observable<number> = from([1, 2, 3]);
const subject = new Subject<number>();

const multicasted = observable.pipe(multicast(subject)) as
   ConnectableObservable<number>;

multicasted.subscribe({
  next: (value) => console.log(`A : ${value}`)
});
multicasted.subscribe({
  next: (value) => console.log(`B : ${value}`)
});
multicasted.connect();
```

Exemple

```
const observable: Observable<number> = from([1, 2, 3]);
const subject = new Subject<number>();

const multicasted = observable.pipe(multicast(subject)) as
   ConnectableObservable<number>;

multicasted.subscribe({
  next: (value) => console.log(`A : ${value}`)
});
multicasted.subscribe({
  next: (value) => console.log(`B : ${value}`)
});
multicasted.connect();
```


Le résultat est

```
// A : 1
// B : 1
// A : 2
// B : 2
// A : 3
// B : 3
```

Angular

Pour se désabonner, on appelle la méthode `unsubscribe()` depuis la souscription

```
ngOnInit() {
    const observable: Observable<number> = interval(1000).pipe(take(10)
      );

    const subject = new Subject<number>();

    const multicasted = observable.pipe(multicast(subject)) as
      ConnectableObservable<number>;

    const a = multicasted.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });

    const b = multicasted.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });

    multicasted.connect();

    setTimeout(() => a.unsubscribe(), 3000);
    setTimeout(() => b.unsubscribe(), 5000);
}
```

Angular

Behavior Subject

- Une des variantes de `Subject` fonctionnant avec la notion de valeur actuelle
- Il stocke la dernière valeur émise par ses observateurs
- Chaque fois qu'un nouvel observateur s'abonne, il reçoit immédiatement la valeur actuelle
- Le constructeur de la classe `BehaviorSubject` prend comme paramètre la valeur initiale

Angular

Exemple

```
ngOnInit() {
    const subject = new BehaviorSubject(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);
}
```

Angular

Exemple

```
ngOnInit() {
    const subject = new BehaviorSubject(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);
}
```


Le résultat est

```
// A : 0
// A : 1
// A : 2
// B : 2
// A : 3
// B : 3
```

Angular

ReplaySubject

- Une des variantes de `Subject` fonctionnant avec la notion de nombre de valeurs à conserver dans l'historique
- Il conserve un nombre de valeurs dans l'historique afin qu'elles puissent être envoyées aux nouveaux abonnés.
- Le constructeur de la classe `ReplaySubject` prend comme paramètre le nombre de valeurs à conserver dans l'historique

Exemple

```
ngOnInit() {
    const subject = new ReplaySubject<number>(2);
    subject.next(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);
}
```

Exemple

```
ngOnInit() {
    const subject = new ReplaySubject<number>(2);
    subject.next(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);
}
```


Le résultat est

```
// A : 0
// A : 1
// A : 2
// B : 1
// B : 2
// A : 3
// B : 3
```

Angular

Async Subject

- Une des variantes de `Subject`
- Il envoie seulement la dernière valeur à ses observateurs et c'est lorsqu'il finit quel que soit l'ordre de leur abonnement
- Pour signaler sa fin, il appelle la méthode `complete`

Angular

Exemple

```
ngOnInit() {
    const subject = new AsyncSubject();

    subject.next(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);

    subject.complete();
}
```

Angular

Exemple

```
ngOnInit() {
    const subject = new AsyncSubject();

    subject.next(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);

    subject.complete();
}
```

Le résultat est

```
// A : 3
// B : 3
```