ADVANCED PROGRAMMING IN ANGULAR

2020.06.07

TABLE OF CONTENT

Pipe

Pipe is a simple way to transform values in Angular template.

A Pipe takes a value(s) and return a value.

There are some built in pipes, but we can build our own pipes easily.

src/app/hero-birthday1.component.ts

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

@Component({
  selector: 'app-hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988
}
```

src/app/app.component.html

```html
<p>The hero's birthday is {{ birthday | date }}</p>
```

Importance of Pipe

Pipe is very useful to perform operations that require state and change detection.

Using Pipe, we can reduce the unnecessary recalculation/data transform on change detection.

Component using Template Function

```
@Component({
  selector: 'app-root',
  template:  `
    Counter: {{counter}} <br>
    Squared: {{square(counter)}} <br>
    <button (click)="increaseCounter()">Increase Counter</button>
  `,
})
export class CounterComponent {
  public counter = 0;

  square(value) {
    return value * value;
  }

  increaseCounter() {
    this.counter += 1;
  }
}
```

Component using Pipe

```
@Pipe({
  name: 'square'
})
export class SquarePipe implements PipeTransform {
  transform(value): any {
    return value * value;
  }
}
```

```
@Component({
  selector: 'app-root',
  template:  `
    Counter: {{counter}} <br>
    Squared: {{counter | square}} <br> //pipe is used here
    <button (click)="increaseCounter()">Increase Counter</button>
  `,
})
export class...
```

Observable & RxJS

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.

Observables

Observables are lazy collections of multiple values over time.

The main differences between a Promise and Observable are as follows:

- A *Promise* is eager, whereas an *Observable* is lazy
- A *Promise* is always asynchronous, while an *Observable* can be either synchronous or asynchronous
- A *Promise* can provide a single value, whereas an *Observable* is a stream of values (from 0 to multiple values)

Eager vs Lazy

```
1  const greetingPoster = new Promise((resolve, reject) => {
2    console.log('Inside Promise (proof of being eager)');
3    resolve('Welcome! Nice to meet you.');
4  });
5
6  console.log('Before calling then on Promise');
7
8  greetingPoster.then(res => console.log(`Greeting from promise: ${res}`));
```

Result:

Inside Promise (proof of being eager)
Before calling then on Promise
Greeting from promise: Welcome! Nice to meet you.

Eager vs Lazy

```
1  const { Observable } = rxjs;
2
3  const greetingLady$ = new Observable(observer => {
4    console.log('Inside Observable (proof of being lazy)');
5    observer.next('Hello! I am glad to get to know you.');
6    observer.complete();
7  });
8
9  console.log('Before calling subscribe on Observable');
10
11 greetingLady$.subscribe({
12   next: console.log,
13   complete: () => console.log('End of conversation with preety lady')
14 });
```

Eager vs Lazy

Result:

Before calling subscribe on Observable
Inside Observable (proof of being lazy)
Hello! I am glad to get to know you.
End of conversation with pretty lady

Common Operators

Area	Operators
Creation	from, fromEvent, of, …
Combination	combineLatest, concat, merge, startWith, withLatestFrom, zip, …
Filtering	debounceTime, distinctUntilChanged, filter, take, takeUntil, …
Transformation	bufferTime, concatMap, map, mergeMap, scan, switchMap, …
Utility	tap, delay, timeout, …
Multicasting	share, …

Most commonly used operators

- map, mergeMap, switchMap
- forkJoin
- combineLatest
- concat
- filter
- catchError

mergeMap

Projects each source value to an Observable which is merged in the output Observable.

Returns an Observable that emits items based on applying a function that you supply to each item emitted by the source Observable, where that function returns an Observable, and then merging those Observables and emitting the results of this merger.

mergeMap

```
1. import { of, interval } from 'rxjs';
2. import { mergeMap, map } from 'rxjs/operators';
3.
4. const letters = of('a', 'b', 'c');
5. const result = letters.pipe(
6.   mergeMap(x => interval(1000).pipe(map(i => x+i))),
7. );
8. result.subscribe(x => console.log(x));
9.
10. // Results in the following:
11. // a0
12. // b0
13. // c0
14. // a1
15. // b1
16. // c1
17. // continues to list a,b,c with respective ascending
    integers
```

switchMap

Projects each source value to an Observable which is merged in the output Observable, emitting values only from the most recently projected Observable.

Returns an Observable that emits items based on applying a function that you supply to each item emitted by the source Observable, where that function returns an Observable. Each time, it observes one of these inner Observables, the output Observable begins emitting the items emitted by that inner Observable. When a new inner Observable is emitted, it stops emitting items from the old emitted inner Observable and begins emitting items from the new one.

switchMap

```
1. import { of } from 'rxjs';
2. import { switchMap } from 'rxjs/operators';
3.
4. const switched = of(1, 2, 3).pipe(switchMap((x: number) =>
   of(x, x ** 2, x ** 3)));
5. switched.subscribe(x => console.log(x));
6. // outputs
7. // 1
8. // 1
9. // 1
10. // 2
11. // 4
12. // 8
13. // ... and so on
```

forkJoin

Accepts an Array of ObservableInput or a dictionary Object of ObservableInput and returns an Observable that emits either an array of values in the exact same order as the passed array, or a dictionary of values in the same shape as the passed dictionary.

It takes any number of input observables which can be passed either as an array or a dictionary of input observables. If no input observables are provided, resulting stream will complete immediately.

forkJoin

```
1. import { forkJoin, of, timer } from 'rxjs';
2.
3. const observable = forkJoin({
4.   foo: of(1, 2, 3, 4),
5.   bar: Promise.resolve(8),
6.   baz: timer(4000),
7. });
8. observable.subscribe({
9.  next: value => console.log(value),
10.  complete: () => console.log('This is how it ends!'),
11. });
12.
13. // Logs:
14. // { foo: 4, bar: 8, baz: 0 } after 4 seconds
15. // "This is how it ends!" immediately after
```

combineLatest

Combines multiple Observables to create an Observable whose values are calculated from the latest values of each of its input Observables.

It combines the values from all the Observables passed as arguments. This is done by subscribing to each Observable in order and, whenever any Observable emits, collecting an array of the most recent values from each Observable. So if you pass n Observables to operator, returned Observable will always emit an array of n values, in order corresponding to order of passed Observables (value from the first Observable on the first place and so on).

combineLatest

```
1. import { combineLatest, of } from 'rxjs';
2. import { delay, starWith } from 'rxjs/operators';
3.
4. const observables = [1, 5, 10].map(
5.   n => of(n).pipe(
6.     delay(n * 1000),   // emit 0 and then emit n after n
   seconds
7.     startWith(0),
8.   )
9. );
10. const combined = combineLatest(observables);
11. combined.subscribe(value => console.log(value));
12. // Logs
13. // [0, 0, 0] immediately
14. // [1, 0, 0] after 1s
15. // [1, 5, 0] after 5s
16. // [1, 5, 10] after 10s
```

concat

Creates an output Observable which sequentially emits all values from given Observable and then moves on to the next.

It joins multiple Observables together, by subscribing to them one at a time and merging their results into the output Observable. You can pass either an array of Observables, or put them directly as arguments. Passing an empty array will result in Observable that completes immediately.

concat

```
1. import { concat, interval } from 'rxjs';
2. import { take } from 'rxjs/operators';
3.
4. const timer1 = interval(1000).pipe(take(10));
5. const timer2 = interval(2000).pipe(take(6));
6. const timer3 = interval(500).pipe(take(10));
7.
8. const result = concat(timer1, timer2, timer3);
9. result.subscribe(x => console.log(x));
10.
11. // results in the following:
12. // (Prints to console sequentially)
13. // -1000ms-> 0 -1000ms-> 1 -1000ms-> ... 9
14. // -2000ms-> 0 -2000ms-> 1 -2000ms-> ... 5
15. // -500ms-> 0 -500ms-> 1 -500ms-> ... 9
```

catchError

Catches errors on the observable to be handled by returning a new observable or throwing an error.

```
import { of } from 'rxjs';
import { map, catchError } from 'rxjs/operators';

of(1, 2, 3, 4, 5).pipe(
    map(n => {
          if (n === 4) {
              throw 'four!';
      }
     return n;
    }),
    catchError(err => of('I', 'II', 'III', 'IV', 'V')),
  )
  .subscribe(x => console.log(x));
  // 1, 2, 3, I, II, III, IV, V
```

Thank You