Creatively Decouple ngOnChanges

A nicer way to subscribe to property changes.

Siyang Kern Zhao Follow
Jan 7 · 3 min read

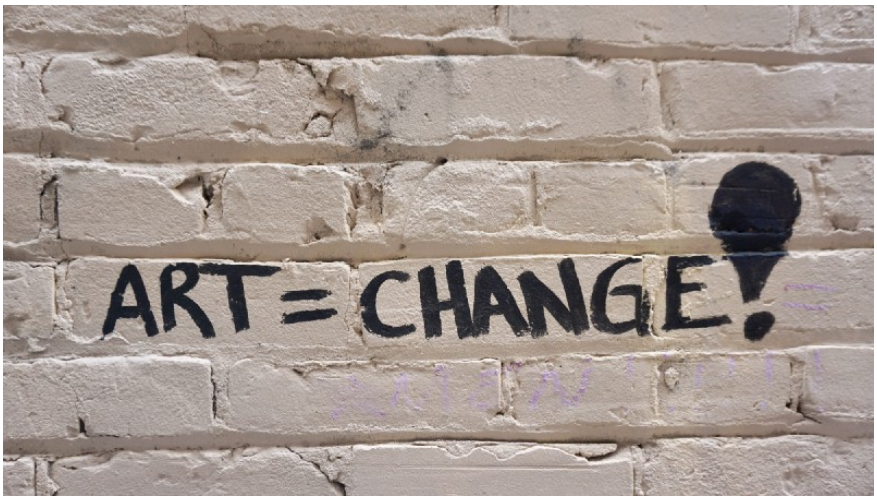

Photo by Peder Cho on Unsplash

When it comes to subscribing to property changes in Angular, I think most people would immediately think of the `ngOnChanges` lifecycle hook. A typical example looks like this:

```
ngOnChanges(changes: SimpleChanges) {
  if (changes.key1) {
      console.log(`key1 is changed from
${changes.key1.previousValue} to
${changes.key1.currentValue}`);
  }
  if (changes.key2) {
    console.log(`key2 is changed from
${changes.key2.previousValue} to
${changes.key2.currentValue}`);
  }
  // ...
}
```

Personally, I am NOT a big fan of `ngOnChanges` for the following reasons:

1. It combines change detection of ALL input properties into one `ngOnChanges` hook function. And then we need to separate those properties with an `if` statement making it less readable especially when there are many properties to be watched.
2. The interface of `SimpleChanges` accepts any string as its key, making it possible for typos. For example, `changes.typo_key` will not be complained about by the TypeScript compiler.
3. `SimpleChange.previousValue` and `SimpleChange.currentValue` are typed to `any` instead of the desired property type.

```
export interface SimpleChanges {
    [propName: string]: SimpleChange;
}
```

A slightly better way (Not my favorite yet)

I have seen a common alternative to `ngOnChanges`, which is to use a setter function. It looks like this:

```
export class AppComponent {
  private _title: string;

  @Input()
  set title(value: string) {
    this._title = value;
    console.log(`title is changed to ${value}`);
  }

  get title(): string {
    return this._title;
  }
}
```

Advantages

1. This decouples the different properties. The setter function (on-change hook) is located together with `@Input()` for better readability.

Disadvantages

1. A “private” ghost property `_title` needs to be created. Furthermore, it is not really “private” as `_title` is still accessible and changeable anywhere inside the component, which is not what we really want. What we wanted is that the title can only be read/written though getter/setter functions. But, this is not enforced.
2. Lengthy code: I just want to subscribe to `title` change, why do I have to bother introducing `_title` and a getter function.

Decorator to the rescue (My favorite)

I am a big fan of TypeScript decorators. They allow us to do a lot of meta-programming nicely.

Blueprint

```
export class AppComponent {
  @OnChange<string>(function (value, simpleChange) {
      console.log(`title is changed to: ${value}`);
  })
  @Input()
  title: string;
}
```

How to implement OnChange

```
// This is different from Angular's SimpleChange as it adds
generic type T
export interface SimpleChange<T> {
  firstChange: boolean;
  previousValue: T;
  currentValue: T;
  isFirstChange: () => boolean;
}
```

```
export function OnChange<T = any>(callback: (value: T,
simpleChange?: SimpleChange<T>) => void) {
  const cachedValueKey = Symbol();
  const isFirstChangeKey = Symbol();
  return (target: any, key: PropertyKey) => {
    Object.defineProperty(target, key, {
      set: function (value) {
        // change status of "isFirstChange"
        if (this[isFirstChangeKey] === undefined) {
          this[isFirstChangeKey] = true;
        } else {
```

```
            this[isFirstChangeKey] = false;
          }
          // No operation if new value is same as old value
          if (!this[isFirstChangeKey] && this[cachedValueKey]
=== value) {
            return;
          }
          const oldValue = this[cachedValueKey];
          this[cachedValueKey] = value;
          const simpleChange: SimpleChange<T> = {
            firstChange: this[isFirstChangeKey],
            previousValue: oldValue,
            currentValue: this[cachedValueKey],
            isFirstChange: () => this[isFirstChangeKey],
          };
          callback.call(this, this[cachedValueKey],
simpleChange);
        },
        get: function () {
          return this[cachedValueKey];
        },
      });
    };
}
```

Advantages

1. Intuitive, easy to use, less code, better readability.
2. As powerful as `ngOnChanges` since `simpleChange` is available
3. Hide `_cachedValue` from developer, no more “ghost property”.
4. Better typing. `SimpleChange.previousValue` is typed to a generic type.
5. It can also be used with a non-`@Input` property.
6. It’s not specific to Angular. So it can be used as long as it’s TypeScript such as React in TypeScript.

Some notes about decorator

The TypeScript decorator is experimental and is bound to change as the specs move along. So, use it with caution.

Try it out

StackBlitz

Edit description

stackblitz.com

Available in npm

property-watch-decorator

A decorator for watching property change

www.npmjs.com