The Ultimate Answer To The Very Common Angular Question: subscribe() vs | async Pipe

Tomas Trajan Follow

Oct 2, 2018 · 7 min read

Original 📷 by louis amal

Most of the popular Angular state management libraries like NgRx expose application state in a form of a stream of state objects. This is usually implemented with the help of RxJS Observables.

> *The state updates get pushed to the components which react by re-rendering their templates to display the most recent version of the*

> *application state.*

There are multiple ways in which it is possible to consume this observable stream of state updates by the components, the two most popular being:

1. Using the `subscribe()` method and storing the state on the component instance, `todos$.subscribe(todos => this.todos = todos)` ...
2. The `| async` pipe unwraps the state object directly in the component's template, `<li *ngFor="let todo of todos$ | async"></li>` ...

> *I have been thinking about this question and related trade-offs for quite some time now. I was mostly in favor of using subscribe() but couldn't really point out exact reasons why.*

This led to a need to come up with an overview of the situation while considering all the pros and cons to create a final guideline to be able to make objective decision in every case!

Before we dig deeper, I would like to thank simply10w and Tim Deschryver who provided lots of great input and feedback (PR) while working on Angular NgRx Material Starter project which implements ideas discussed in this article...

Topics

Please notice following things which make a big impact on the final answer to this question...

- the way `| async` pipe works for the collection vs for singular objects
- possibility of using new-ish `*ngIf` "as" syntax (from Angular 4 and above)
- location of state handling logic (component's class vs template)

> *OK, let's do this!*

Case 1: Use subscribe() in the ngOnInit method

```
@Component({
  /* ... */
  template: `
    <ul *ngIf="todos.length > 0">
      <li *ngFor="let todo of todos">{{todo.name}}</li
    </ul>
  `
})
export class TodosComponent implements OnInit, OnDestr
  private unsubscribe$ = new Subject<void>();

  todos: Todo[];

  constructor(private store: Store<State>) {}

  ngOnInit() {
    this.store
```

Simple example of consuming observable stream of todos by unwrapping it in the component **ngOnInit** method and using unwrapped property in the template

👍 Advantages of using subscribe()

1. Unwrapped property can be used in multiple places in the template “as it is” without the need to rely on various workarounds as will be shown in the case 2 examples

2. Unwrapped property is available everywhere in the component. This means it can be **used directly in the component’s method** without the need to pass it from the template. That way, all state can be kept in the component.

```
toggleAllTodosToDone() {
  // unwrapped todos accessible directly in component's method
  this.todos.forEach(todo =>
    this.store.dispatch(new ActionToggleTodo({ id: todo.id, done: true })));
}
```

The only responsibility of the template is to trigger this method (eg using **(click)="toggleAllTodosToDone()"**)

👎 Disadvantages of using subscribe()

1. Using of the `subscribe()` introduces complementary need to unsubscribe at the end of the component life-cycle to avoid memory leaks. Developers usually have to unsubscribe manually. The most RxJS (declarative) way to do this is to employ `takeUntil(unsubscribe$)` operator as shown in the example above. This solution is verbose and error prone because it is very easy to forget implementing `ngOnDestroy` which will **not** lead to any errors just a silent memory leak...

2. Subscribing to the observable manually in the `ngOnInit()` doesn't work with `OnPush` change detection strategy out of the box. We could make it work by using `this.cd.markForCheck()` inside of our subscribe handler but this is a very easy to forget, error prone solution.

```
constructor(
  private store: Store<State>,
  private cd: ChangeDetectorRef                          // inject change detector
) {}

ngOnInit() {
  this.store
    .pipe(select(selectTodos), takeUntil($unsubscribe)) // unsubscribe to prevent memory leak
    .subscribe(todos => {
      this.todos = todos;                                // unwrap observable
      this.cd.markForCheck();                            // trigger change detection
    });
}
```

The issue with the OnPush change detection strategy was the final straw and the deal-breaker for my previously favorite

"subscribe()" approach to handling of the observable data sources in the Angular components

. . .

Case 2: Use | async pipe in the component template

```
@Component({
  /* ... */
  template: `
    <ul *ngIf="(todos$ | async).length">
      <li *ngFor="let todo of todos$ | async">{{todo.n
    </ul>
  `
})
export class TodosComponent implements OnInit {
  todos$: Observable<Todo[]>;

  constructor(private store: Store<State>) {}
```

Simple example of consuming observable stream of todos using | async pipe in the components template

👍 Advantages of using | async pipe

1. Solution **works with** `OnPush` change detection out of the box! Just make sure that all your business logic (eg reducer, service) is immutable and always returns new objects. Anyway, this is the whole purpose of using NgRx in a first place so I guess immutable data can be assumed...

2. Angular handles subscriptions of `| async` pipes for us automatically so there is no need to unsubscribe manually in the component using `ngOnDestroy`. This leads to less verbosity and hence less possibilities for making a mistake. Yaaay 😸

Follow me on Twitter to get notified about the newest Angular blog posts and

interesting frontend stuff!👀

👎 Disadvantages of using | async pipe

1. Objects have to be unwrapped in the template using `*ngIf="something$ | async as something"` syntax. On the other hand, this is not a problem with collections which get unwrapped in a straight forward manner when using `*ngFor="let something of somethings$ | async"`.

2. Objects have to be potentially unwrapped multiple times in a single template in multiple different places. This can be avoided by using a dedicated wrapper element but that means that the state management is mandating changes to `DOM` structure which is pretty weird...

3. Properties unwrapped in the template using `*ngIf` or `*ngFor` are **not** accessible in the component's methods. This means we have to pass these properties to the methods from the template as the method parameters which further splits logic between the template and the component itself...

```
<h1>{{(something | async).title}}</h1>        <!-- multiple | async pipes -->
<p>{{(something | async).description}}</p>
<ul>
  <li *ngFor="let item of (something | async).items">{{item.name}}</li>
</ul>

<!-- versus -->

<div *ngIf="something$ | async as something"> <!-- wrapper element -->
  <h1>{{something.title}}</h1>
  <p>{{something.description}}</p>
  <ul>
    <li *ngFor="let item of something.items">{{item.name}}</li>
  </ul>
</div>
```

Multiple subscriptions using | async pipes versus dedicated wrapper element with ***ngIf "as"** syntax

We could also use `<ng-container>` instead of `<div>`. This way, no new wrapper element will be created in the final `DOM` but we still end up with wrapped element in the template source code.

Many thanks to Martin Javinez for suggesting solution with multiple `| async` pipes resolved into one variable...

Many thanks to Ward Bell for pointing out that besides using wrapper `<ng-container></ng-container>` element there is an another way of preventing multiple subscription by multiple | async pipes in our templates...

The solution is to pipe our observable stream through `ReplaySubject` like this `something$ = sourceOfSomething$.pipe(ReplaySubject(1));`

. . .

The Verdict 🥊

The `OnPush` change detection strategy is great for performance so we should be using `| async` pipe as much as possible. More so, we could argue that the consistent use of these features simplifies application because developers can always assume one way data flow and automatic subscription management.

> *The | async pipe is the clear winner*

Considerations

The `subscribe()` solution **does** provide some benefits in terms of template readability and simplicity. It can be the right solution in case we're not using `OnPush` change detection strategy and are not planing to use it in the future...

Also, check out related great tip from Tim Deschryver. If we find ourselves in a situation when our template is getting too complex and bloated with many `| async` pipes unwrapping a lot of objects we should consider breaking that component into a smart wrapper and multiple dumb view components...

After that you can simply pass unwrapped property inside your dumb component like this `<dumb [prop]="value$ | async"></dumb>` so you have working `OnPush` while having a benefit of working with the unwrapped objects in the potentially complex template of the dumb component.

```
@Component({ // smart (container) component
  /* ... */
  template: `<todo-list [todos]="todos$ | async"></todo-list>` // dumb component consumes unwrapped todos
})
export class TodosComponent implements OnInit {
  todos$: Observable<Todo[]>;

  constructor(private store: Store<State>) {}

  ngOnInit() {
    this.todos$ = this.store.pipe(select(selectTodos))
  }
}


@Component({ // dumb (view) component
  /* ... */
  template: `
    <ul>
      <li *ngFor="let todo of todos">{{todo.name}}</li>
    </ul>
  `
})
export class TodoList {
  @Input() todos: Todo[];
}
```

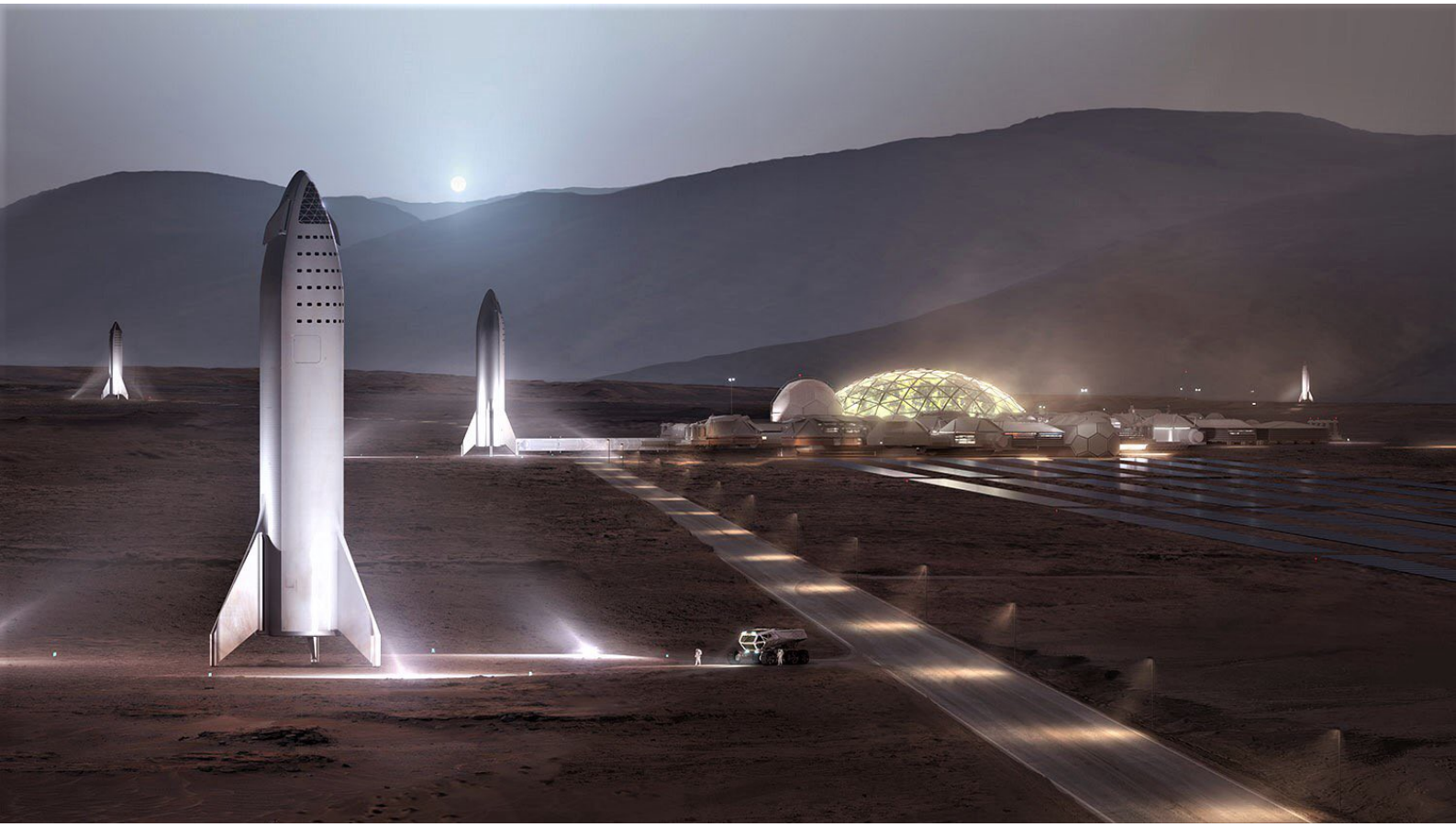
In case of complex template it often make sense to extract parts of it into stand-alone dumb components which can work directly with unwrapped objects while benefiting from the **OnPush** change detection strategy

That's It For Today!

> *I hope you enjoyed this article and will now have a mental model that helps you to decide whether to use* `subscribe()` *or* `| async` *pipe in the components of your Angular applications!*

Please support this guide with your 👏👏👏 and help it spread to a wider audience 🙏. Please, don't hesitate to ping me if you have any questions using the article responses or Twitter DMs @tomastrajan

And never forget, future is bright



Obviously the bright future (📷 by SpaceX)

If you made it this far, feel free to check out some of my other articles about Angular and frontend software development in general...

💁‍♂️ The 7 Pro Tips To Get Productive With Angular CLI & Schematics 💪

Angular Schematics is a workflow tool for the modern web—official introduction...

medium.com

Total Guide To Angular 6+ Dependency Injection—providedIn vs providers:[ ] 💉

Let's learn when and how to use new better Angular 6+ dependency injection...

medium.com

How To Stay Up To Date With Releases Of Popular Frameworks

Introducing Release Butler—A Twitter Bot That Helps You To Stay Up To Date With...

medium.com

How To Build Responsive Layouts With Bootstrap 4 and Angular 6 📐

Every web app is assumed to be responsive, period.

medium.com

Total Guide To Dynamic Angular Animations That Can Be Customized At...

Animations make projects look much better

blog.angularindepth.com

How Did Angular CLI Budgets Save My Day And How They Can Save Yours

Budgets is one of the less known features of the Angular CLI which helps you to kee...

medium.com