ANGULAR UNIVERSITY

PREMIUM QUALITY
ANGULAR TUTORIALS

angular-university.io

Angular OnPush Change Detection and Component Design - Avoid Common Pitfalls

Did you ever try to use the Angular OnPush Change Detection strategy in your application, but run into some hard to debug issues and quickly went back to default change detection?

In this post we are going to cover some typical pitfalls where OnPush is giving unexpected results and how to fix those situations, we will see that OnPush is a lot simpler to use than it might look at first sight and is compatible with all sorts of component designs.

If you are also looking to learn more about the default change detection mechanism, have a look at this post How does Angular Change Detection Really Work ?

Scenario 1 - Our Starting Point (default change detection)

Let's have a look at a simple component that does not use yet OnPush change detection, it's a newsletter component: we will use it in a parent `HomeComponent` that looks like the following:

```

@Component({
  selector: 'home',
  template: `

  <newsletter [user]="user" (subscribe)="subscribe($event)"></newsletter>
```

```

    <button (click)="changeUserName()">Change User Name</button>

`})
export class HomeComponent {

    user: User = {
        firstName: 'Alice',
        lastName: 'Smith'
    };

    constructor(private newsletterService: NewsletterService) {

    }

    subscribe(email:string) {
        this.newsletterService.subscribe(email);
    }

    changeUserName() {
        this.user.firstName = 'Bob';
    }

}

```

As we can see, we are passing the `User` data as an input. `User` is a simple custom type defined as the following:

```

export interface User {
    firstName:string;
    lastName:string;
}

```

As we can see on the `Home` component, the newsletter component is receiving as input a reference to a user object, which is currently hard-coded at the level of the `Home` component.

There is also a button "Change User Name", that will mutate the user data directly.

Initial implementation of the newsletter component

The newsletter component is currently written as a purely presentational component, that simply takes inputs and display them on the template, and emits an `@Output` event when the subscription occurs:

```

@Component({
    selector: 'newsletter',
    template: `

<fieldset class="newsletter">

    <legend>Newsletter</legend>

    <h5>Hello {{user?.firstName}},
        enter your email below to subscribe:</h5>

    <form>

        <input #email type="email" name="email" placeholder="Enter your Ema

        <input  type="button" class="button button-primary" value="Subscrib
                (click)="subscribeToNewsletter(email.value)">
    </form>

</fieldset>

`})
export class NewsletterComponent  {

    @Input()
    user: User;

    @Output()
    subscribe = new EventEmitter();

    subscribeToNewsletter(email:string) {
        this.subscribe.emit(email);
    }

}


```

As we can see the newsletter component takes the user object as an `@Input()`, and displays the first name in the template.

Default Change Detection and Object Mutability

If we test this example by clicking in the "Change User Name" button, everything will work as expected, meaning that:

- initially, the text inside the newsletter will say "Hello Alice", because that was the value defined in the `Home` component
- when we click on the change name button, the text will now say "Hello Bob", because that is the value directly set on the `changeUserName` method

This implementation with direct mutability of the user data works because we are using the Angular default change detection mechanism, which is compatible with direct object mutation.

Angular will compare the result of the expression `{{user?.firstName}}` before and after the click event, a change will be detected and the template will be updated with the new value.

But what if we use OnPush change detection instead?

OnPush change detection and Direct Object mutability

Let's change the newsletter component so that it uses OnPush change detection:

```

@Component({
    selector: 'newsletter',
    changeDetection: ChangeDetectionStrategy.OnPush,
    template: `...`
})
export class NewsletterComponent  {
 ....
}

```

If we now push again on the "Change Name" button, the text inside the newsletter component will remain as "Hello Alice", so our application is giving incorrect results - the view does not reflect the model anymore.

Why isn't this working with OnPush?

So far so good, we have an error scenario but we were actually expecting this situation - there are other scenarios below that are likely less familiar.

This situation occurs because:

- we mutated the user object directly
- but OnPush works by comparing references of the inputs of the component
- because we did not provide a reference to a new object but instead mutated an existing one, the OnPush change detector

did not get triggered

Avoiding direct object mutation with OnPush

If we would change the implementation of `changeUserName()` to create a new user instance instead of mutating the existing instance, everything would work as expected:

```

changeUserName() {
    this.user = {
        firstName: 'Bob',
        lastName: 'Smith'
    }
}

```

With this version of `changeUserName()` and OnPush, the text would now be "Hello Bob" after clicking the "Change User Name" button.

To avoid this issue, we simply need to either avoid mutating objects directly or use an immutability library to freeze the view model data that we pass to our components.

Actually so far this is how we would expect OnPush to work: but there is more to it than meets the eye. Even being aware of this you would still run into situations where OnPush seems like it's not working.

OnPush change detection and event handlers

Is there any other way that OnPush could know that the component needs to be re-rendered? Notice that inside the newsletter component there is a button with a click handler.

If we click on the "Subscribe" button, we will see that now the template shows "Hi Bob", so the triggering of event handlers inside the component itself also causes the on push change detector to trigger, independently than if the inputs have changed or not.

So this our first indication that OnPush is more than about checking input properties.

Are there more scenarios where OnPush is also triggered?

Scenario 2 - OnPush and Observables

Let's say that now the user data is not hard-coded at the level of the user component.

To make it a more realistic scenario, let's say that this data is available at a centralized `UserService`, that loads the data at startup time and makes the data available to any part of the application via dependency injection.

Let's have a look at what the user service looks like:

```

const ANONYMOUS_USER: User = {
    firstName: '',
    lastName: ''
};


@Injectable()
export class UserService {

   private subject = new BehaviorSubject<User>(ANONYMOUS_USER);

   user$: Observable<User> = this.subject.asObservable();
```

```

    loadUser(user:User) {
        this.subject.next(user);
    }

}

```

This is a simplified implementation of what a service like this would look like, normally this service would retrieve the user data from the backend using another service.

The UserService implementation

Let's break down what is going on at the level of this service:

- this is a global singleton service because we will add it to our root application module
- The user data is available via a publicly available observable named `user$`, to which other parts of the application can subscribe to
- the observable is derived from a private subject instance using `asObservable`
- the service emits new values of data via a private subject, to which other parts of the application cannot access
- by not exposing the subject, the `UserService` controls the ability to emit new user objects to other parts of the application
- the service initially emits an `ANONYMOUS_USER` object but also exposes a public method `loadUser`
- `loadUser` normally would for example call the backend and load the data from there, but in this case and for

demonstration purposes, we will simply emit a new user directly

Receiving user data in other parts of the application

Let's now inject this service for example in Home component, and see how to use it:

```

@Component({
    selector: 'home',
    template: `

    <newsletter [user]="userService.user$ | async"
        (subscribe)="subscribe($event)"></newsletter>

    <button (click)="changeUserName()">Change User Name</button>

`})
export class HomeComponent {

    constructor(
        private newsletterService: NewsletterService,
        private userService: UserService ) {
    }

    subscribe(email:string) {
        this.newsletterService.subscribe(email);
    }

    changeUserName() {
        this.userService.loadUser({firstName: 'Bob', lastName: 'Smith' });
    }
}

```

So let's break it down the changes we made here:

- we have injected the `UserService`
- we have consumed the `userService.user$` observable directly in the home component using the async pipe

Scenario 4 results

If we test now this application, what is the result now if we click the "Change User Name" button, will the name inside the newsletter change?

> *With this implementation of the home component that uses the observable user service, everything is still working correctly*

meaning that the text on the screen would now be "Hello Bob".

Why is this new version still working?

This is because we have emitted a new user object via the observable, so from the point of view of the newsletter component a new user object instance is still being received, so everything still works.

Scenario 3 - Passing Observables as @Inputs() to a OnPush component

But now we would like to change a bit the design of our application. Let's say that instead of subscribing to the `user$` observable directly at the level of the home component, we would like to pass this observable to the component tree:

```

@Component({
    selector: 'home',
    template: `

    <newsletter [user$]="userService.user$" (subscribe)="subscribe($event)"
    </newsletter>

    <button (click)="changeUserName()">Change User Name</button>

`})
export class HomeComponent {

    ....

}
```

As we can see, everything in the Home component remains the same but now we are passing a reference to the `user$` observable to the newsletter component.

This reference will always be the same as we emit new values of this observable.

How to handle the observable at the level of the child component?

Let's now see what the newsletter component would look like:

```
@Component({
    selector: 'newsletter',
    changeDetection: ChangeDetectionStrategy.OnPush,
    template: `
<fieldset class="newsletter">

    <legend>Newsletter</legend>

    <h5>Hello {{(user$ | async).firstName}},
        enter your email below to subscribe:</h5>

     ...

</fieldset>
`
})
export class NewsletterComponent  {

    @Input()
    user$: Observable<User>;

    ...

}

```

As we can see, the component now has an input property which is an Observable, that gets subscribed to via the async pipe.

Scenario 3 results - is this still working with OnPush?

So what will happen in this case? In this case, there are no changes in the input property `user$`. It's still referencing the same object that just happens to be an observable.

So based on the previous error scenario, because the input property did not change we could think that the template would not be updated:

> *But that is not case, Scenario 3 is still working correctly !*

The `user$` observable is being subscribed to via the async pipe, so Angular knows that the emission of values in that observable will impact the template.

And this is why with version 3, if we click on "Change User Name" the newsletter template will reflect the change and everything is still working correctly.

Let's now try other application designs and see if we run into change detection issues while using OnPush change detection.

Scenario 4 - Deeply nested smart components

But now we decided to do a refactoring: we now want to nest the newsletter component much deeper inside the component tree. And the tree of components contains third party libraries for which we don't necessarily have the code.

We would like to build the newsletter component so that it takes all the data that it needs from services instead of inputs, to avoid having to:

- pass data up the component tree without the intermediary components needing the data (avoiding extraneous inputs)
- manually bubbling custom `Output()` events down the component tree to parent components, repeating the event in each intermediary component

Bubbling events manually several levels up the component tree is really inconvenient and is a likely sign that the component design needs to be revisited.

Let's then re-implement the newsletter component as a deeply nested smart component:

```

@Component({
    selector: 'newsletter',
    changeDetection: ChangeDetectionStrategy.OnPush,
    template: `

    <fieldset class="newsletter">
```

```

        <legend>Newsletter</legend>

        <h5>Hello {{firstName}},
            enter your email below to subscribe:</h5>

        <form>
            <input #email type="email" name="email" placeholder="Enter your

            <input  type="button" class="button button-primary" value="Subs
                    (click)="subscribeToNewsletter(email.value)">
        </form>

    </fieldset>

`})
export class NewsletterComponent implements OnInit {

    firstName:string;

    constructor(
        private newsletterService: NewsletterService,
        private userService: UserService) {

    }

    ngOnInit() {
        this.userService.user$.subscribe(
          user => this.firstName = user.firstName
        );
    }

    subscribeToNewsletter(email:string) {
        this.newsletterService.subscribe(email);
    }
}

```

What does the Home component look like now?

So now this component can be injected anywhere inside the Home component sub-tree, and it will still work. In this case the home component would now look like this:

```
@Component({
    selector: 'home',
    template: `

    ... deep tree of components that include the newsletter component ...

    <button (click)="changeUserName()">Change User Name</button>

`})
export class HomeComponent {

    constructor(private userService: UserService) {

    }

    changeUserName() {
        this.userService.loadUser({firstName: 'Bob', lastName: 'Smith'});
    }

}
```

This is an example of how sometimes its better to inject services deeply in the component tree instead of passing data and bubbling events up and down the component tree, and the dependency injection system makes it really practical to do that.

There is only one problem with this approach:

> *this implementation of the Newsletter component does not work with OnPush change detection !*

Why is OnPush not working for this component ?

This implementation manually subscribes to the `user$` observable in `ngOnInit` will only work with the default change detection mechanism, but not with OnPush.

Does this mean that with OnPush we cannot deeply inject services in our component tree ? No, we simply need to make sure that any observables that we inject directly via constructor services are subscribed to at the template level using the async pipe:

```

@Component({
    selector: 'newsletter',
    changeDetection: ChangeDetectionStrategy.OnPush,
    template: `

    <fieldset class="newsletter" *ngIf="userService.user$ | async as user e

        <legend>Newsletter</legend>

        <h5>Hello {{user.firstName}},
            enter your email below to subscribe:</h5>

        <form>
            <input #email type="email" name="email" placeholder="Enter your

            <input  type="button" class="button button-primary" value="Subs
                    (click)="subscribeToNewsletter(email.value)">
        </form>

    </fieldset>

    <ng-template #loading>
        <div>Loading ...</div>
    </ng-template>

`})
export class NewsletterComponent  {

    constructor(
        private newsletterService: NewsletterService,
        private userService: UserService) {

    }

    subscribeToNewsletter(email:string) {
        this.newsletterService.subscribe(email);
    }

}

```

Scenario 4 fixed - deeply nested smart component working with OnPush

This implementation now works great with OnPush change detection !

This is because we have subscribed to the `user$` using the async pipe, so now the OnPush change detector of the newsletter can be triggered each time `user$` emits a value.

But before using the async pipe, there was no way for the framework to know that values emitted by this observable where being passed the template.

Some new features introduced in Angular 4

Notice also a couple of new things:

- we are using the new `ngIf` 'as' syntax (introduced in Angular 4) to assign the output of the async pipe to a template variable named `user`
- we are using the new `ngIf` 'else' syntax to show a loading indicator while the data is not yet available

Conclusions

As we can see, if we take some precautions in the way we build our components, OnPush will work transparently with all sorts of component designs - components that receive data directly as inputs, that have observable inputs, or components that receive data only via constructor services, etc.

An OnPush change detector gets triggered in a couple of other situations other than changes in component `Input()` references, it also gets triggered for example:

- if a component event handler gets triggered
- if an observable linked to the template via the async pipe emits a new value

So if we remember to subscribe to any observables as much as possible using the async pipe at the level of the template, we get a couple of advantages:

- we will run into much less change detection issue using OnPush
- we will make it much easier to switch from the default change detection strategy to OnPush later if we need to
- Immutable data and `@Input()` reference comparison is not the only way to achieve a high performant UI with OnPush: the reactive approach is also an option to use OnPush effectively