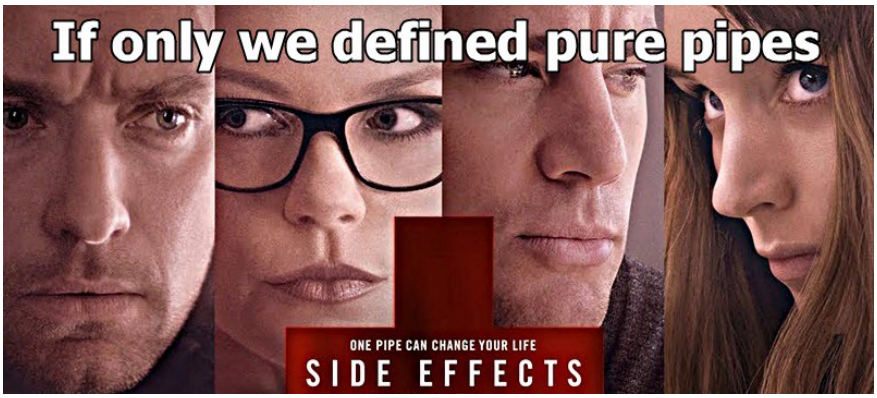
The essential difference between pure and impure pipes in Angular and why that matters

Max Koretskyi aka Wizard Follow
Sep 1, 2017 · 7 min read



> **We're organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I'll be happy to see you at the conference. We'll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

When writing a custom pipe in Angular you can specify whether you define a pure or an impure pipe:

```
@Pipe({
  name: 'myCustomPipe',
  pure: false/true        <----- here (default is `true`)
})
export class MyCustomPipe {}
```

Angular has a pretty good documentation on pipes that you can find here. But as it often happens with documentation the clearly

reasoning for division is missing. In this article I'd like to fill that hole and demonstrate the difference from the prospective of functional programming which shows where the idea of pure and impure pipes come from. Besides learning the difference you will know *how* it affects the performance and this knowledge will help you write efficient and performant pipes.

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide "**Get started with Angular grid in 5 minutes**". I'm happy to answer any questions you may have. **And follow me to stay tuned!**

• • •

A pure function

There's so much information on the web about functional programming that probably every developer knows what a pure function is. For myself I define a pure function as a function that doesn't have an internal state. It means that all operations it performs are not affected by that state and given the same input parameters and produces the same deterministic output.

Here are the two versions of a function that adds numbers. The first one is pure and the second one is impure:

```
const addPure = (v1, v2) => {
  return v1 + v2;
};

const addImpure = (() => {
  let state = 0;
  return (v) => {
    return state += v;
  }
})();
```

If I call both functions with the same input, say number 1, the first one will produce the same output `2` on every call:

```
addPure(1, 1);  // 2
addPure(1, 1);  // 2
```

```
addPure(1, 1);  // 2
```

while the second one produces different output:

```
addImpure(1);  // 1
addImpure(1);  // 2
addImpure(1);  // 3
```

So the key takeaway here is that even if the input doesn't change the impure function can produce different output. **It means that we cannot use the input value to determine if the output will change.**

Let's see another interesting consequence of the fact that a function has a state. Suppose you have a `calculator` object that takes as a parameter the function to add numbers and uses it to make calculations:

```
class Calculator {
  constructor(addFn) {
    this.addFn = addFn;
  }

  add(v1, v2) {
    return this.addFn(v1, v2);
  }
}
```

If the function is pure and doesn't have a state it can be safely shared with many instances of the `Calculator` classes:

```
class Calculator {
  constructor(addFn) {
    this.addFn = addFn;
  }

  add(v1, v2) {
    return this.addFn(v1, v2);
  }
}

const c1 = new Calculator(add);
const c2 = new Calculator(add);
```

```
c1.add(1, 1); // 2
c2.add(1, 1); // 2
```

However, **the function that is not pure can't be shared**. This is because the operations performed by one instance of `Calculator` will affect the function state and consequently the result of operations performed by the other instance of `Calculator`:

```
const add = (() => {
  let state = 0;
  return (v) => {
    return state += v;
  }
})();

class Calculator {
  constructor(addFn) {
    this.addFn = addFn;
  }

  add(v1, v2) {
    return (this.addFn(v1), this.addFn(v2));
  }
}

const c1 = new Calculator(add);
const c2 = new Calculator(add);

c1.add(1, 1); // 2
c2.add(1, 1); // 4  <------ here we have `4` instead of `2`
```

You can see that the first call to `add` method on the second instance produces the output `4` instead of expected `2`.

So let's recap what we've learnt so far about functions:

Pure:

- input parameters value determine the output so if input parameters don't change the output doesn't change
- can be shared across many usages without affecting the output result

Impure:

- cannot use the input value to determine if the output will change

- cannot be shared because the internal state can be affected from outside

• • •

Applying that knowledge to Angular pipes

Suppose we defined one custom pipe and make it pure:

```
@Pipe({
  name: 'myCustomPipe',
  pure: true
})
export class MyCustomPipe {}
```

And use it like this in a component template:

```
<span>{{v1 | customPipe}}</span>
<span>{{v2 | customPipe}}</span>
```

Since the pipe is pure it means that there's no internal state and the pipe can be shared. How can Angular leverage that? Even though there are two usages in the template Angular can **create only one pipe instance which can be shared between the usages**. For those who know from my previous articles what a component factory is here is the relevant compiled code which defines only one pipe definition:

```
function View_AppComponent_0(_l) {
  return viewDef_1(0, [
    pipeDef_2(0, ExponentialStrengthPipe_3, []), // node
index 0
    ...
```

which is shared in updateRenderer function:

```
function(_ck,_v) {
    unwrapValue_7(_v,4,0,_ck(_v,5,0,nodeValue_8(_v, 0),...);
                                                    ^^^
    unwrapValue_7(_v,8,0,_ck(_v,9,0,nodeValue_8(_v, 0),...);
                                                    ^^^
```

Here is the `unwrapValue` function is used to retrieve the current pipe value by calling `transform` on it. The pipe instance is referenced by the node index in the `nodeValue` function call—which is `0` in this case.

However, if we define the pipe as impure assuming there's some internal state:

```
@Pipe({
  name: 'myCustomPipe',
  pure: false
})
export class MyCustomPipe {}
```

We don't want the pipe in the second usage to be affected by the call to it in the first usage so angular **creates two instances** of the pipe each with its own state:

```
function View_AppComponent_0(_l) {
   return viewDef(0, [
       ...
       pipeDef_2(0, ExponentialStrengthPipe, []) // node
index 4
       ...
       pipeDef_2(0, ExponentialStrengthPipe, []) // node
index 8
```

and it is **not shared** in updateRenderer function:

```
function(_ck,_v) {
    unwrapValue_7(_v,4,0,_ck(_v,5,0,nodeValue_8(_v, 4),...);
                                                    ^^^
    unwrapValue_7(_v,8,0,_ck(_v,9,0,nodeValue_8(_v, 8),...);
                                                    ^^^
```

You can see here that instead of node index `0` now for each usage Angular uses different node index— `4` and `8` respectively.

The second point from the summary in the first chapter was that with pure functions we can use the input value to determine if the output will change while with impure functions we can't have such guarantee.

In Angular we pass input parameters to a pipe like this:

```
<span>{{v1 | customPipe:param1:param2}}</span>
```

So if a pipe is pure we know that it's output (through transform method) is strictly determined by the input parameters. If the input parameters don't change the output won't change. This reasoning allows Angular to optimize the pipe and call `transform` method **only when input parameters change**.

But if a pipe is impure and has internal state the same parameters do not guarantee that same output as demonstrated with the call to impure `addFn` function in the first chapter. It means that Angular is forced to trigger `transform` function on a pipe instance on every digest.

A good example of impure pipe is the AsyncPipe from `@angular/common` package. This pipe has internal state that holds an underlying subscription created by subscribing to the observable passed to the pipe as a parameter. Because of that Angular has to create a new instance for each pipe usage to prevent different observables affecting each other. And also has to call `transform` method on each digest because even thought the observable parameter may not change the new value may arrive through this observable that needs to be processed by change detection.

The other two impure pipes are JsonPipe and SlicePipe. Angular puts an additional restriction on a pipe to be considered pure—the input to the pipe cannot be mutable. If the input is mutable, a pipe need to be re-evaluated on every digest because an input object can be mutated without changing the object reference (the pipe parameter stays the same). And that's exactly why both JsonPipe and SlicePipe pipes are not considered pure despite not having an internal state.

The rest Angular default pipes are pure.

. . .

Conclusion

So as we've seen impure pipes can have significant performance hit if not used wisely and carefully. The performance hit comes from the fact that Angular creates multiple instances of an impure pipe and also calls it's `transform` method on every digest cycle.

I hope by reading the article you now know the difference between the two types, how Angular handles both of them and what mental model should you use when designing and implementing your custom pipes.

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

For more insights follow me on Twitter and on Medium.