



How to write good, composable and pure components in ANGULAR



Jacek Tomaszewski

Fullstack Web Developer at [Recruitee.com](https://recruitee.com)

Freelancer and Consultant at jtom.me

ng-poznan #31 @ Poznań,
5th June 2018



Problem

Writing new components is easy...
... but modifying and extending them is **not**.

(Especially when you have hundreds of them,
and you want to avoid breaking things.)



Problem

Whatever framework you're using (Angular / React / Vue?), all the time the same exact questions tend to appear again:

- Where do I keep the state of X?
- How do I change the state of X?
- If the state of X depends on Y, how do I wire up those dependencies?
- If I can solve a problem P with either solution A, B or C, which one should I pick?

Real world example a search page

Where should we keep
the state?

How should we split the
components?

The screenshot shows a web interface for managing candidates. The top navigation bar is blue and contains a menu icon, a search bar, and user profile information. The sidebar on the left lists candidate status filters: 'Qualified candidates' (42), 'New candidates' (84), 'Not contacted' (84), and 'Followed candidates' (0). Below these are filters for 'Candidate status' (Qualified (29), Disqualified (42), New (63)) and 'In Job' status. The main content area is titled 'Qualified candidates' and shows a list of 29 candidates. The table columns are Name, Evaluation, Job, Stage, and Date. The candidates listed are Jerry Thompson, Boris Spencer, Bria Willmes, Leora Ferry, Zachery Bahring, Harry Roob, Sally Glover, and Darryl Teunissen.

Name	Evaluation	Job	Stage	Date
<input type="checkbox"/> Jerry Thompson	—	● Manager (example)	Interview	15d ago
<input type="checkbox"/> Boris Spencer	—	● Designer (example)	Phone screen	16d ago
<input type="checkbox"/> Bria Willmes	—	● Designer (example)	Interview	17d ago
<input type="checkbox"/> Leora Ferry	—	● Designer (example)	Phone screen	17d ago
<input type="checkbox"/> Zachery Bahring	—	● Manager (example)	Applied	20d ago
<input type="checkbox"/> Harry Roob	—	● Designer (example)	Offer	21d ago
<input type="checkbox"/> Sally Glover	—	● Manager (example)	Evaluation	23d ago
<input type="checkbox"/> Darryl Teunissen	—	● Manager (example)	Phone screen	23d ago

Solution

1. Divide components into Smart and Dumb
2. Keep components as Dumb as possible
3. Decide when a component should be Smart instead of Dumb



1. Divide components into Smart and Dumb

Dumb Component

Also known as:

- Pure Component
- Presentational Component

Smart Component

Also known as:

- Impure Component
- Connected Component
- Container Component

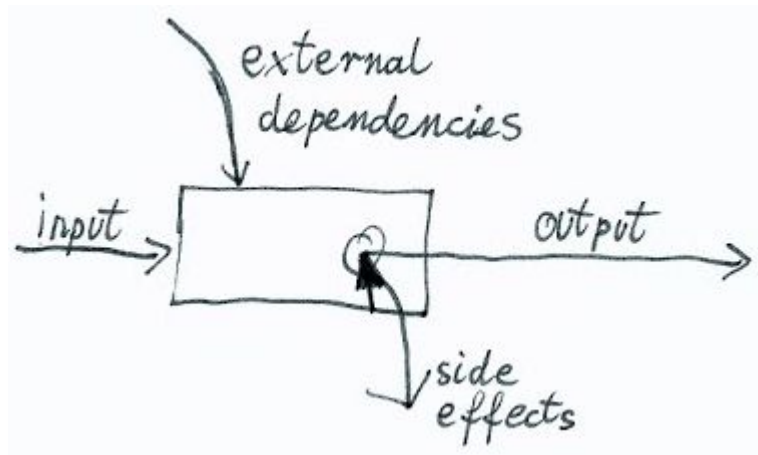
Dumb Component

- **is like a pure function:** a function that for given function arguments, will always produce the same return value;
- for given data (via inputs) always looks and behaves the same;
- emits events (via outputs).



Smart Component

- **is like an impure function:** a function that touches "the outer world": either by getting data from external services or by producing side effects;
- receives data from external services;
- produces side effects in external services.





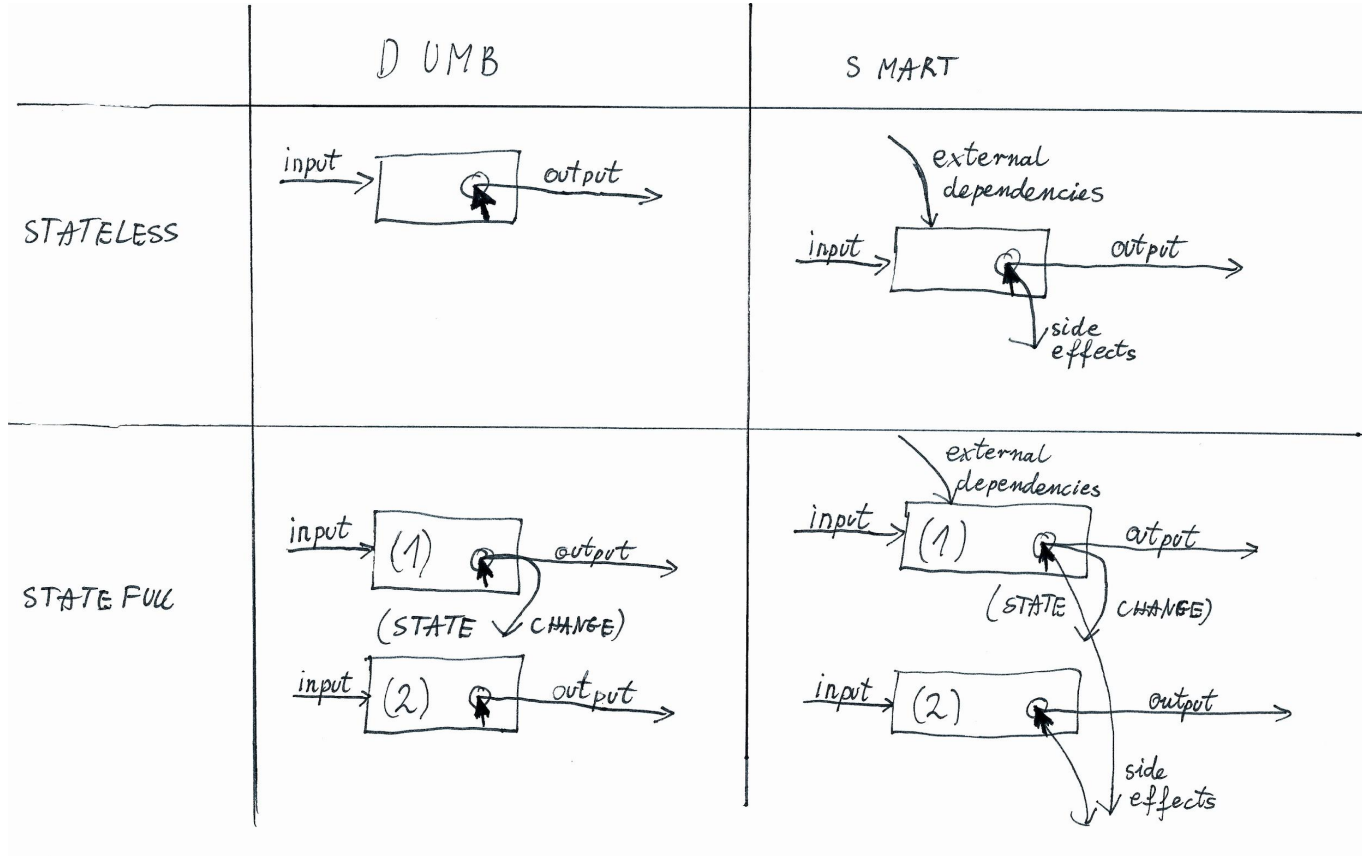
Note: Smart/Dumb is not Stateful/Stateless!

A **Dumb Component** has no external dependencies and causes no side effects.

A **Smart Component** has external dependencies or causes side effects.

A **Stateless Component** has no local state
- but might still cause side effects.

A **Stateful Component** has a local state
- but he doesn't need to have any dependencies nor cause any side effects.




Smart/Dumb x Stateful/Stateless matrix



2. Keep components as Dumb as possible

- a. Avoid making them dependant on external services
- b. Avoid producing any kind of side effects
- c. Never mutate @Input() data




Avoid making them dependant on external services

If it needs any data to work, inject it via `@Input()`.

```
// wrong (impure)
class DateTimePickerComponent {
    timeZone: string = 'Europe/Warsaw';


    constructor(
        private account: UserAccount
    ) {
        if (this.account.currentUser) {
            this.timeZone =
this.account.currentUser.timeZone;
        }
    }
}
```



Avoid making them dependant on external services

If it needs any data to work, inject it via
`@Input()`.

```
// good (pure)
class DateTimePickerComponent {
    @Input() timeZone: string =
    'Europe/Warsaw';
}
```



Avoid making them dependant on external services

If it needs any data to work, inject it via `@Input()`.

```
// ok (pure and impure mix)

class DateTimePickerComponent {

    @Input() timeZone: string =
    'Europe/Warsaw';


    constructor(
        @Optional() private account:
        UserAccount
    ) {

        if (this.account &&
            this.account.currentUser) {

            this.timeZone =
            this.account.currentUser.timeZone;

        }

    }
}
```



Avoid making them dependant on external services

If it needs any data to work, inject it via `@Input()`.

```
// wrong (impure)

class MessageItemComponent {

    @Input() message: MessageData;

    messageUnread: boolean = false;

    constructor(

        private messages: MessagesRepo,
        private cdRef: ChangeDetectorRef
    ) {}

    ngOnInit() {

        this.messages

            .isMessageUnread(this.message.id)

            .subscribe(unread => {

                this.messageUnread = unread;

                if (!this.cdRef['destroyed']) {


                    this.cdRef.detectChanges();

                }

            });

    }

}
```



Avoid making them dependant on external services

If it needs any data to work, inject it via `@Input()`.

```
// good (pure)
class MessageItemComponent {
    @Input() message: MessageData;
    @Input() messageUnread: boolean =
false;
}
```




Avoid producing any kind of side effects

If it needs to emit something, emit it through `@Output()`.

```
// wrong (impure)

class DateTimePickerComponent {
  timeZone: string = 'Europe/Warsaw';

  changeTimeZone(timeZone: string) {
    this.timeZone = timeZone;
    this.accountRepo.updateCurrentUser({
      timeZone
    });
  }
}
```



Avoid producing any kind of side effects

If it needs to emit something, emit it through `@Output()`.

```
// good (pure)

class DateTimePickerComponent {
    @Input() timeZone: string =
    'Europe/Warsaw';

    @Output() timeZoneChange:
    EventEmitter<string> = new
    EventEmitter();

    changeTimeZone(timeZone: string) {
        this.timeZoneChange.emit(timeZone);
    }
}
```



Avoid producing any kind of side effects

If it needs to emit something, emit it through `@Output()`.

```
// wrong (impure)

class MessageItemComponent {
    @Input() message: MessageData;
    messageUnread: boolean = false;

    markMessageAsRead(read: boolean) {
        this.messageUnread = !read;
        this.messagesRepo.markMessageAsRead({
            id: message.id,
            read: read
        });
    }
}
```



Avoid producing any kind of side effects

If it needs to emit something, emit it through `@Output()`.

```
// good (pure)

class MessageItemComponent {

    @Input() message: MessageData;

    @Input() messageUnread: boolean =
false;

    @Output() messageMarkedAsRead:
EventEmitter<boolean> = new
EventEmitter();

    markMessageAsRead(read: boolean) {
        this.messageMarkedAsRead.emit(read);
    }
}
```



Never mutate @Input() data

If it needs to change something, emit the change as an event, which the parent will then pick up and properly act on it.

(Because mutating @Input() data is actually a production of a side effect that causes a change in the parent component's data.)

```
class MessageItemComponent {  
  @Input() message: MessageData;  
  
  get messageUnread(): boolean {  
    return this.message._isUnread;  
  }  
  
  markMessageAsRead(read: boolean) {  
    // suuper wrong  
    // - never do it please!  
    this.message._isUnread = !read;  
  
    this.messagesRepo.markMessageAsRead({  
      id: message.id,  
      read: read  
    });  
  }  
}
```



3. Decide when a component should be Smart

- a. If it can be Dumb, make it Dumb.
- b. If multiple children are equally Smart, make them Dumb.
- c. What cannot be Dumb, make it Smart.
- d. If the Smart one gets too big, divide it into separate Smarts.



If it can be Dumb, make it Dumb

TextInputComponent:
receives a text value and
eventually emits a valueChange
event

```
@Component(...)  
class TextInputComponent {  
    @Input() value: string;  
    @Output() valueChange: EventEmitter<string>;  
}
```



If it can be Dumb, make it Dumb

DateRangeFilterComponent:
receives a date range filter value
and eventually emits a value
change event

```
@Component(...)
class DateRangeFilterComponent {
    @Input() value: DateRangeFilterValue;
    @Output() valueChange:
    EventEmitter<DateRangeFilterValue>;
}
```




If it can be Dumb, make it Dumb

Note: If it needs to inject and use an external JS library to show a date picker - it is not a problem!

If the Dumb component's behaviour is consistent according to its' Inputs, and it doesn't cause side effects that might alter our main app's behaviour, then it's a fine Dumb component as well.

DateRangeFilterComponent:
receives a date range filter value
and eventually emits a value
change event

```
@Component(...)
class DateRangeFilterComponent {
    @Input() value: DateRangeFilterValue;
    @Output() valueChange:
    EventEmitter<DateRangeFilterValue>;
}
```



If it can be Dumb, make it Dumb

MessageFormComponent:
receives initial message data,
current form loading/failure
indicators and errors, and
eventually emits a submit event

```
@Component(...)
class MessageFormComponent {
    @Input() initialMessage: MessageData;
    @Input() saveLoading: boolean = false;
    @Input() saveFailed: boolean = false;
    @Input() errors?: string[];
    @Output() submit: EventEmitter<MessageData>;
}
```



If it can be Dumb, make it Dumb

Note: See how explicitly we are injecting the most important business logic to it with Inputs and Outputs (saving the form and the result of it).

It is not the Dumb's responsibility to save the form nor update its' state. In this case, it does keep the local state of the message before submitting it, but sending the form and handling the save response will be done elsewhere.

MessageFormComponent:
receives initial message data,
current form loading/failure
indicators and errors, and
eventually emits a submit event

```
@Component(...)
class MessageFormComponent {
    @Input() initialMessage: MessageData;
    @Input() saveLoading: boolean = false;
    @Input() saveFailed: boolean = false;
    @Input() errors?: string[];
    @Output() submit: EventEmitter<MessageData>;
}
```



If multiple children are equally Smart, make them Dumb

That is: if they have the same external dependency/side effects, let's move it to their Smart parent instead, and make the children Dumb instead.

1. For example, if a Search Page has multiple filters like:
 - UserFilterComponent
 - AdminFilterComponent
 - DateRangeFilterComponent

Then instead of having a common external dependency on ``SearchPageService.currentFilters`` in all of them, let's just create a "FiltersComponent", that takes over this dependency and make all sub-filter components beneath it Dumb.



What cannot be Dumb, make it Smart

For example: the top view components.

Because even if we make everyone Dumb, there needs to be at least someone who is Smart.

(The one, who actually saves the data to LocalStorage, obtains the data from APIs, listens to the navigation changes, etc.)



If the Smart one gets too big, divide it into separate Smarts

“Make the top view component Smart” is a very good rule, but in some of the views, it might not be enough.

For example, a Gmail main inbox page has following features:

- lists recent threads in the given folder (and lets you act on them)
- lists available folders (and lets you modify them)
- shows up online people from Hangout
- allows you to quickly write a new message



If the Smart one gets too big, divide it into separate Smarts

If we would have only one Smart component on the Gmail page, then it would need to take over all of its' responsibilities. Quite a lot. How is that better than a spaghetti code?

Instead, we could split into a few Smart Components, that have their own responsibilities:

- `ThreadsListView`
- `FoldersListView`
- `HangoutPeopleListView`
- `NewMessageModalView`

Real world example

Recruitee's search page

[\(live demo\)](#)

The screenshot displays the Recruitee interface for managing candidates. The top navigation bar includes a search bar, a 'Careers Site' link, and several utility icons. A left sidebar provides filters for candidate status and job roles. The main content area shows a list of 29 qualified candidates, each with a profile picture, name, evaluation status, job title, stage, and time since last activity.

Qualified candidates

CANDIDATES (29) NOTES (0)

Search:

1 - 29 of 29 [+ Add candidates](#) [Date](#) [Filter](#) [More](#)

	Name	Evaluation	Job	Stage	Date
<input type="checkbox"/>	Jerry Thompson	—	● Manager (example)	Interview	15d ago
<input type="checkbox"/>	Boris Spencer	—	● Designer (example)	Phone screen	16d ago
<input type="checkbox"/>	Bria Willmes	—	● Designer (example)	Interview	17d ago
<input type="checkbox"/>	Leora Ferry	—	● Designer (example)	Phone screen	17d ago
<input type="checkbox"/>	Zachery Bahringer	—	● Manager (example)	Applied	20d ago
<input type="checkbox"/>	Harry Roob	—	● Designer (example)	Offer	21d ago
<input type="checkbox"/>	Sally Glover	—	● Manager (example)	Evaluation	23d ago
<input type="checkbox"/>	Darryl Teunissen	—	● Manager (example)	Phone screen	23d ago

[+ Add filter](#) [Clear](#)

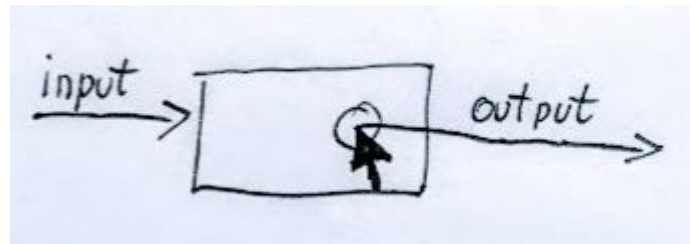


Pros & Cons of the Smart/Dumb split

- + You can easily predict the Dumb Component's behaviour.
- + You can easily test the Dumb Component's behaviour.
- + You can (quite) easily change the Dumb Component's behaviour without breaking things.
- + The main logic of your app is controlled only by your Smart Components.
- + It is more performant.
- + It helps you avoid bugs.
- You cannot inject dependencies wherever you want.
- You cannot mutate data passed through Input/Output.

You can easily predict the Dumb Component's behaviour

Just look at it.



Its' behaviour is as simple as it public Inputs and Outputs interface.

(Which, by the way, together with TypeScript typedefs, serves as an awesome documentation.)




You can easily test the Dumb Component's behaviour

Testing a Dumb Component is as simple as:

1. Define input values
2. Instantiate Component
3. Act on the Component (f.e. click it)
4. Assert that a specific `@Output()` had been emitted.

(Testing a Smart Component usually requires much more than that: stubbing external dependencies, checking for side effects, etc.)




You can (quite) easily change the Dumb Component's behaviour without breaking things

Whenever you change a Dumb Component:


- Make sure the old interface is still working (or search & replace old usages of this Component, which you can easily do thanks to TypeScript)
- The main behaviour of Component still works as intended.

You don't need to see if any external dependencies break this component, or if it produces some other side effects than before. It never did so and never will.



The main logic of your app is controlled only by your Smart Components

- You don't need to read your whole code repository, just to see who's fetching what where and what is changed what and where.
(Because what is going on deep down the tree, you can see just by looking at the HTML template.)
- If you need to alter the main logic of your app, often you don't need to touch the Dumb Components at all (or, the only thing that you need to change in them, are their input values.)



It is more performant

Because now you know what exactly depends on what, you don't need anymore the `NgZone` nor the magical `ChangeDetection` mechanism that rechecks for changes of everything in everywhere:

You can just skip `NgZone` and use `ChangeDetectionStrategy.OnPush` in all of your Dumb Components.



It helps you avoid bugs

- Less coupling of your code;
Splitting it into smaller, more SOLID-like and pure bits;
Avoiding side effects
- all of that decreases the complexity of your code and at the same time decreases the chance that bugs are going to happen in your code.
- Since most of your app depends on typed Inputs and Outputs interfaces now, it decreases the chance of bugs caused by typos and wrongly passed data types a lot.



Bonus: Forget about ``markForCheck()``

... because there's no need to use it anymore!

We don't need to inform the Angular that something had been changed in the parent component, because as its' child component, we are never directly changing it anymore.

(If we do so, we do it through ``@Output()``, and it is the parent that handles the change itself.)



Related links

- ["Mastering the Angular performance - by dropping the magic of Change Detector" - Jack Tomaszewski, medium.com](#)
- ["Code is not art, it's engineering" - Jack Tomaszewski, medium.com](#)
- ["Question: How to choose between Redux's store and React's state?"](#) and Dan Abramov's response to it (creator of Redux)
- ["Presentational and Container Components"](#) - Dan Abramov, medium.com

Thanks!

Any questions?



Jacek Tomaszewski

Fullstack Web Developer at [Recruitee.com](https://recruitee.com) (We're hiring!)

Freelancer and Consultant at jtom.me