

---

# Make your components pure and dumb (and composable)



**Jacek Tomaszewski**

Fullstack Web Developer at [Appunite.com](https://appunite.com)

Freelancer and Consultant at [jtom.me](https://jtom.me)

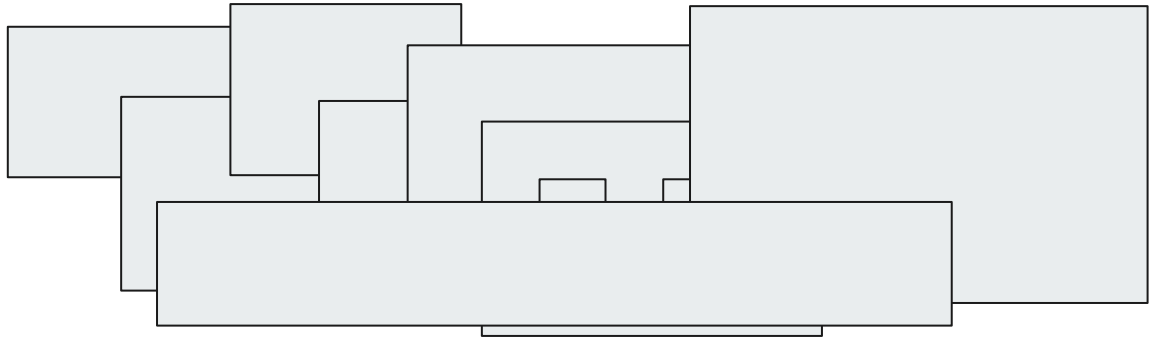
Writing new components is easy...

... but maintaining, modifying and extending them is **not**.



## Writing new components is bad

- leads to having hundreds of components
- decreases readability of the code
- is the opposite to DRY





## Modifying existing components is hard

- risks breaking existing parts of the app
- leads to confusing and complicated code, with lots of `if` and `else` statements
- demands predicting everything in advance

How to make components reusable, simple and useful  
in all the different contexts of the app?

Make them **pure** and **dumb**

# Real world example Search page

The screenshot displays a careers site interface with a blue header. On the left, a sidebar lists candidate categories: Qualified candidates (42), New candidates (84), Not contacted (84), and Followed candidates (0). Below this is a 'Candidate status' filter section with 'Qualified (29)' selected, and an 'In Job' section with an 'Add Job' dropdown. At the bottom of the sidebar are '+ Add filter' and 'Clear' buttons. The main content area is titled 'Qualified candidates' and shows 'CANDIDATES (29)' and 'NOTES (0)'. It includes a search bar, a dropdown menu, and a pagination indicator '1 - 29 of 29'. A '+ Add candidates' button and a 'Date' filter are also present. The main area contains a table with columns for Name, Evaluation, Job, Stage, and Date. The table lists eight candidates with their respective details.

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

---

# Plan

1. Divide components into Smart and Dumb
2. Keep components as Dumb as possible
3. Decide when a component should be Smart, not Dumb





# 1. Divide components into **Dumb** and **Smart**

## Dumb Component

Also known as:

- Pure Component
- Presentational Component

## Smart Component

Also known as:

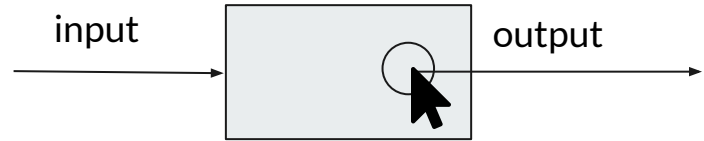
- Impure Component
- Connected Component
- Container Component

---

## Dumb Component

**Pure function** = a function, that for given arguments, always returns the same value.

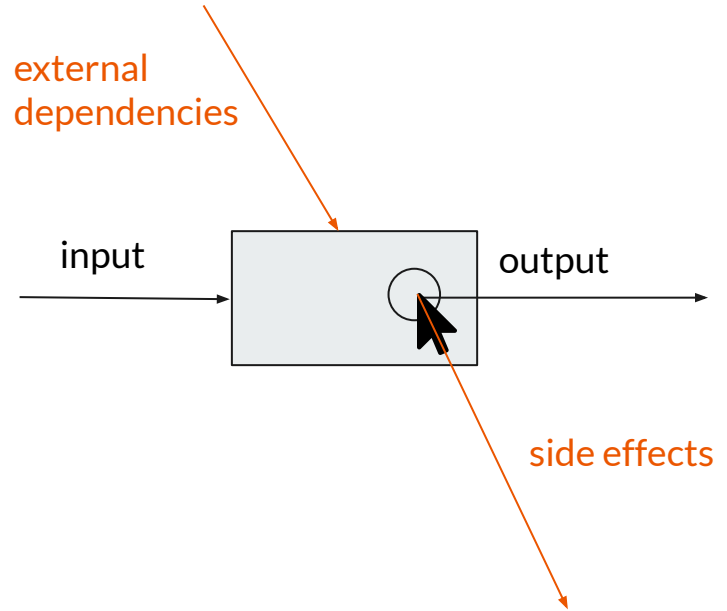
1. For given data (inputs), always looks and behaves the same.
2. Emits events (outputs).



# Smart Component

**Impure function** = a function, that by touching "the outer world", modifies it, and/or returns different result for the same arguments.

1. Receives data from external services.
2. Produces side effects.





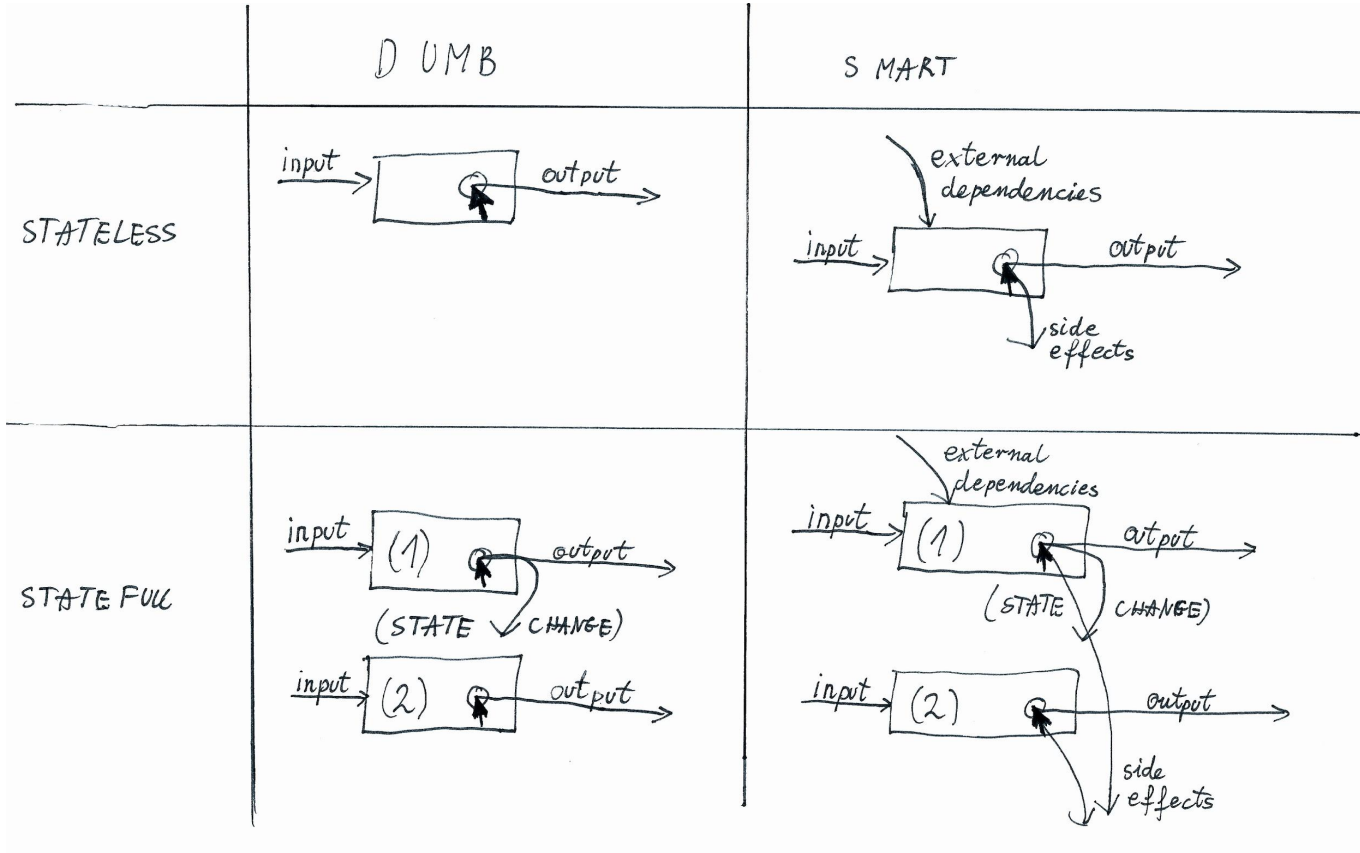
## Note: **Dumb/Smart** is not **Stateless/Stateful!**

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

**Stateless Component** has no local state (but might still cause side effects).

**Smart Component** has external dependencies and/or causes side effects.

**Stateful Component** has a local state (but may not require any external dependencies nor cause any side effects).



Dumb/Smart x Stateful/Stateless matrix



## 2. Keep components as **Dumb** as possible

- a. Don't depend on external services
- b. Don't produce side effects
- c. Don't mutate inputs



# Don't depend on external services

If you need data,  
require it directly.

```
class DateTimePickerComponent {  
    timeZone: string = 'Europe/Warsaw';  
  
    constructor(  
        private account: UserAccount  
    ) {  
        if (this.account.currentUser) {  
            this.timeZone =  
                this.account.currentUser.timeZone;  
        }  
    }  
}
```



# Don't depend on external services

If you need data,  
require it directly.

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





# Don't depend on external services

If you need data,  
require it directly.

```
class MessageItemComponent {  
  @Input() message: MessageData;  
  messageUnread: boolean = false;  
  
  constructor(  
    private messages: MessagesRepo,  
  ) {}  
  
  ngOnInit() {  
    this.messages  
      .isMessageUnread(this.message.id)  
      .subscribe(unread => {  
        this.messageUnread = unread;  
      });  
  }  
}
```



# Don't depend on external services

If you need data, require it directly.

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



# Don't produce side effects

Instead, emit an event.

```
class DateTimePickerComponent {  
    timeZone: string = 'Europe/Warsaw';  
  
    constructor(  
        private accountRepo: AccountRepo,  
    ) {}  
  
    changeTimeZone(timeZone: string) {  
        this.timeZone = timeZone;  
        this.accountRepo.updateCurrentUser({  
            timeZone  
        });  
    }  
}
```



# Don't produce side effects

Instead, emit an event.

```
class DateTimePickerComponent {  
    @Input() timeZone: string =  
    'Europe/Warsaw';  
    @Output() timeZoneChange:  
    EventEmitter<string> = new  
    EventEmitter();  
  
    changeTimeZone(timeZone: string) {  
        this.timeZoneChange.emit(timeZone);  
    }  
}
```



# Don't produce side effects

Instead, emit an event.

```
class MessageItemComponent {  
  @Input() message: MessageData;  
  messageUnread: boolean = false;  
  
  constructor(  
    private messages: MessagesRepo,  
  ) {}  
  
  markMessageAsRead(read: boolean) {  
    this.messageUnread = !read;  
    this.messagesRepo.markMessageAsRead({  
      id: message.id,  
      read: read  
    });  
  }  
}
```



# Don't produce side effects

Instead, emit an event.

```
class MessageItemComponent {  
    @Input() message: MessageData;  
    @Input() messageUnread: boolean =  
false;  
    @Output() messageMarkedAsRead:  
EventEmitter<boolean> = new  
EventEmitter();  
  
    markMessageAsRead(read: boolean) {  
        this.messageMarkedAsRead.emit(read);  
    }  
}
```



# Don't mutate inputs

Instead, emit the change as an event, and let the parent pick it up and act upon it.

(Mutating input = producing a side effect that causes mutation of parent's data)

```
class MessageItemComponent {
  @Input() message: MessageData;

  get messageUnread(): boolean {
    return this.message._isUnread;
  }

  markMessageAsRead(read: boolean) {
    this.message._isUnread = !read;

    this.messagesRepo.markMessageAsRead({
      id: message.id,
      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

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

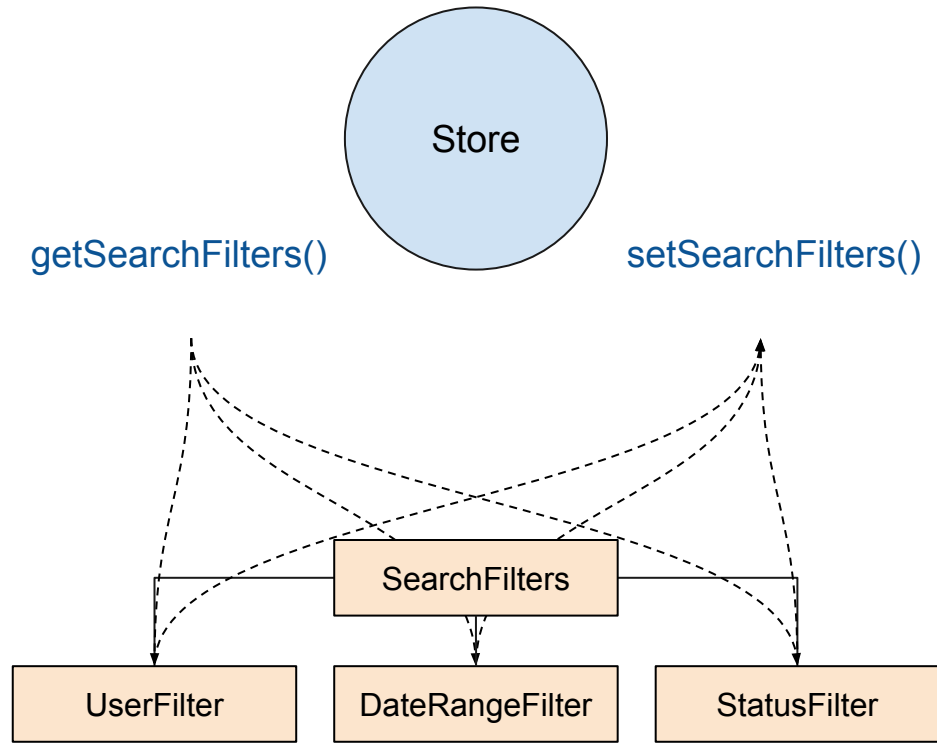


# If it can be Dumb, make it Dumb

```
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

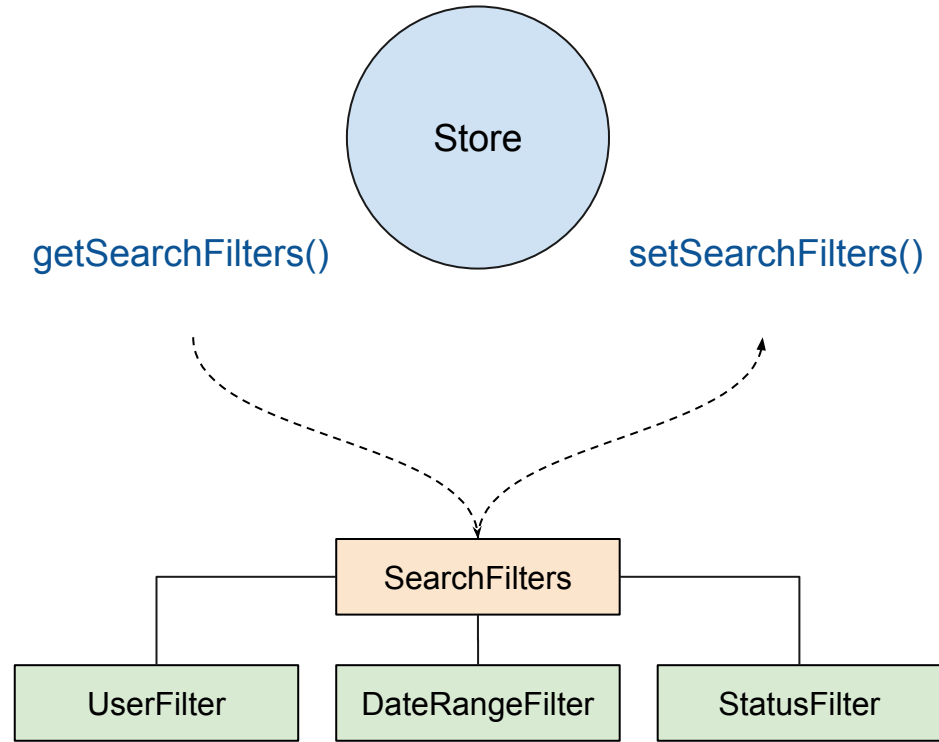
(By moving their shared Smart  
dependency to the parent.)



(bad: everything is Smart)

# If multiple children are equally Smart, make them Dumb

(By moving their shared Smart dependency to the parent.)

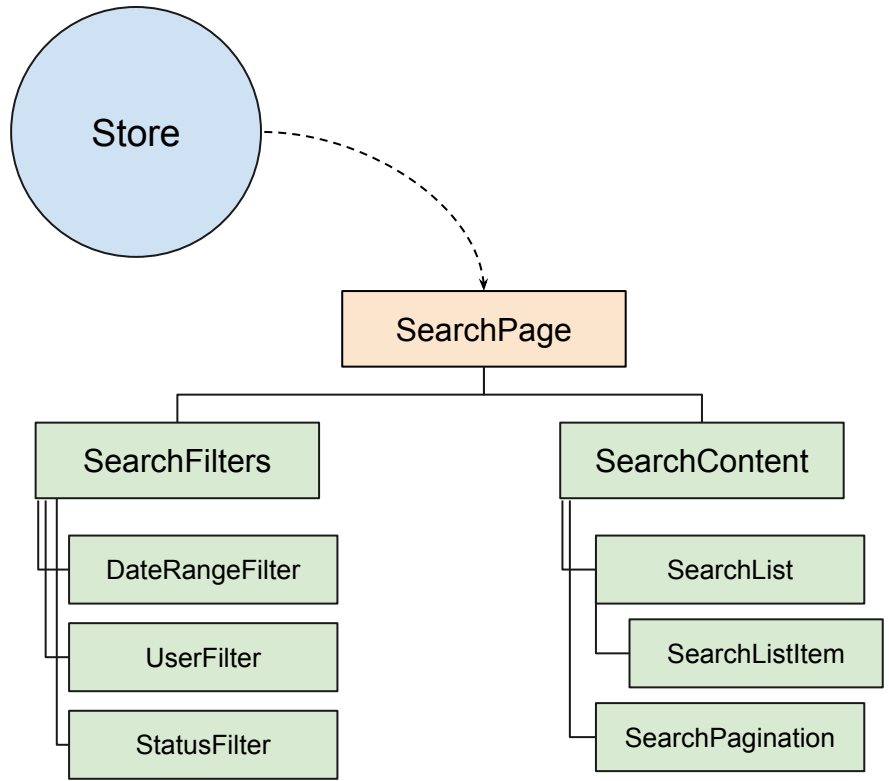


(good: only the parent is Smart)



# What can't be Dumb, make it Smart

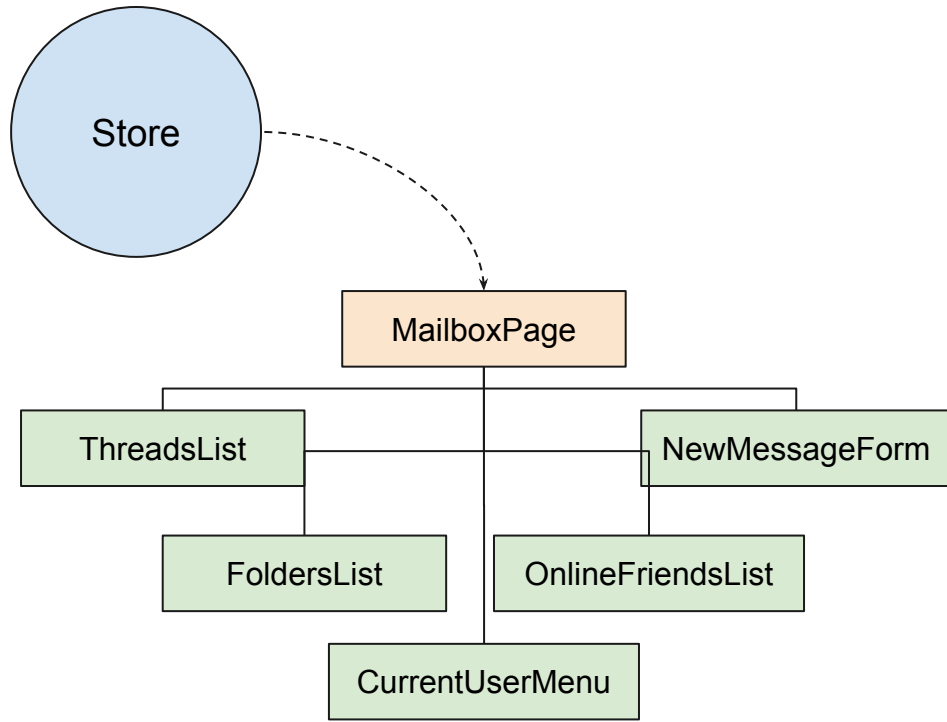
(When everyone is Dumb,  
there needs to be  
at least one Smart.)



—

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

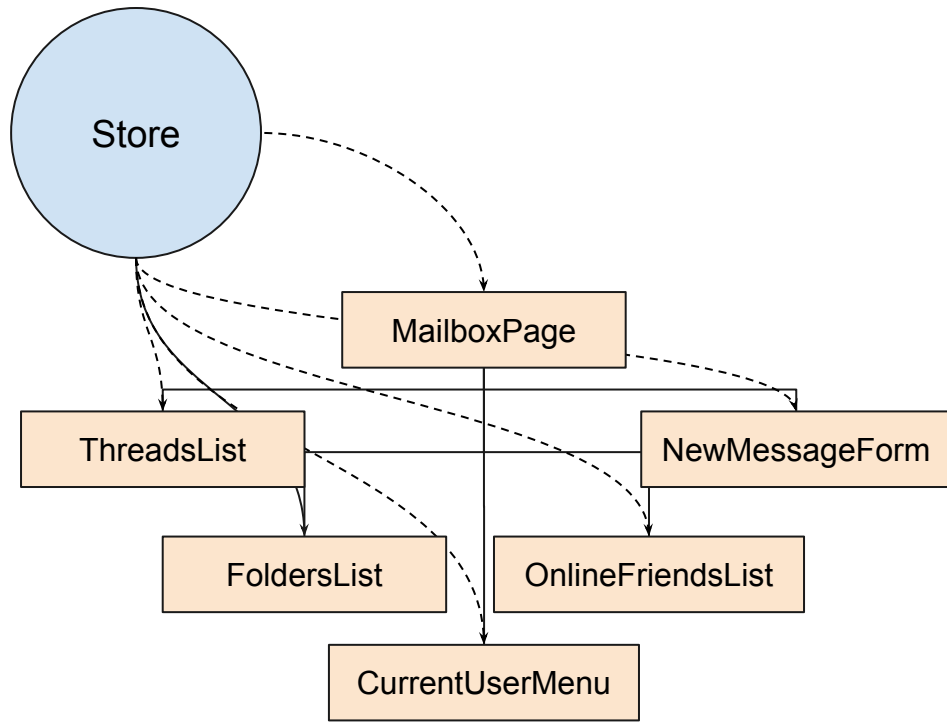
(No one wants a “know-it-all” Smarty Pants.)



—

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

(No one wants a “know-it-all” Smarty Pants.)



# Real world example Search page

The screenshot displays a web interface for managing candidates. At the top, there is a blue header with a search bar labeled "Quick search" and a user profile icon "ST". On the left, a sidebar lists candidate categories: "Qualified candidates" (42), "New candidates" (84), "Not contacted" (84), and "Followed candidates" (0). Below this, a "Candidate status" filter is active for "Qualified (29)", with other options for "Disqualified (42)" and "New (63)". An "In Job" filter is also present. At the bottom of the sidebar are buttons for "+ Add filter" and "Clear".

The main content area is titled "Qualified candidates" and shows "CANDIDATES (29)" and "NOTES (0)". It includes a search bar, a dropdown menu, and a pagination indicator "1 - 29 of 29". A blue button "+ Add candidates" and a "Date" filter are also visible. Below this is a table of candidate details.

	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





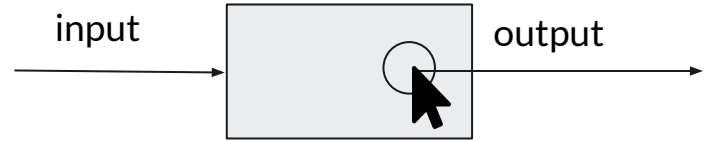
# Pros & Cons of the Smart/Dumb split

- + 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 **Smart** Components.
- + It is more performant.
- + It helps you avoid bugs.
- You cannot inject dependencies wherever you want.
- You cannot mutate data passed through props.

---

# You can easily predict the Dumb Component's behaviour

It's that simple:



(+ with TypeScript typedefs, almost no  
documentation will be needed.)




# 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, just make sure that:


- The old interface is still working (or search & replace all existing usages of this Component)
- 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

If your component is dependant **only** on its' inputs, then you can easily avoid rerendering it if the inputs didn't change.

Less rerenders & view checks -> Win



# 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.



## Related links

- [“How to write good, composable and pure components in Angular 2+” - 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! **Questions?**



**Jacek Tomaszewski**

Fullstack Web Developer at [Appunite.com](https://appunite.com)

Freelancer and Consultant at [jtom.me](https://jtom.me)