title:

RENDERING, REACTIVITY & OBSERVABLES

date:

May 2021

author:

Joan Rieu
@joanrieu
joanrieu.com

HOW TO DISPLAY ANYTHING IN A WEB BROWSER

IMMEDIATE MODE

vs

RETAINED MODE

Two ways of doing the same thing.

IMMEDIATE MODE – STATIC

It’s HTML in its simplest form:

```
<div>Hello world</div>
```

It is displayed by the browser exactly as you wrote it.

RETAINED MODE – STATIC

It’s how the DOM APIs work.

Start by creating an element:

```
const div = document.createElement("div");
```

Set the initial value of the element.

```
div.innerText = "Hello world";
```

Mount the element on the page:

```
document.body.appendChild(div);
```

RETAINED MODE – DYNAMIC

You retain the reference to the `<div>`.

To update the text:

```
div.innerText = "Hello" + name;
```

To unmount the component:

```
div.remove();
```

IMMEDIATE MODE – DYNAMIC

It’s the API offered by React.

```
function Greeting({ name }) {
  return <div>Hello {name}</div>;
}
```

IMMEDIATE MODE – THE GOOD ❤️

Expressive & concise = high signal-to-noise ratio.

It prevents a class of bugs related to bad clean up of previous state.

IMMEDIATE MODE – THE BAD 😔

Sometimes you need to break the abstraction and use the retained-mode API.

It introduces a class of bugs related to bad understanding of the component’s lifecycle.

IMMEDIATE MODE – THE UGLY 👺

It is slow by default.

> *You've changed the tiniest bit of data?*
> *Let's re-render the whole UI!*

MAKING REACT AS FAST AS RETAINED MODE

In three simple steps.

(1/3) NOT RECREATING DOM ELEMENTS

If a DOM element has already been created during a previous render, we want to update it, not delete and recreate it.

So a diff is computed between the output of a new render and the previous one.

PS: This is not only good for performance but also mandatory for correct behaviour of inputs.

(2/3) NOT MAKING UNNECESSARY UPDATES TO THE DOM

> *One ~~train~~* render *may hide another.*

Since sometimes renders are triggered multiple times in a row. Costly DOM updates are avoided by using a Virtual DOM.

- N successive renders
- N virtual DOM updates (fast)
- 1 real DOM update (slow)

(3/3) NOT RENDERING IF PROPS HAVE NOT CHANGED

This is not enabled by default in React. 😨

You need to use `React.memo()` to activate it.

Without it, you are always rendering the whole UI, even if only to Virtual DOM.

This is the most important optimization!

OOPS!

You app is now so optimized that it never re-renders.

Even when it should.

Thanks JavaScript!

JAVASCRIPT’S DEFINITION OF EQUALITY

Quiz time! 🏅

Equal or not equal

```
const a = 1;
const b = 1;
```

`a === b` ?

Equal or not equal

```
const a = 1;
const b = 1;
```

`a === b` ?

YES

Equal or not equal

```
const a = undefined;
const b = null;
```

`a === b` ?

Equal or not equal

```
const a = undefined;
const b = null;
```

`a === b` ?

NO

Equal or not equal

```
const a = {};
const b = a;
```

`a === b` ?

Equal or not equal

```
const a = {};
const b = a;
```

`a === b` ?

YES

Equal or not equal

```
const a = {};
const b = {};
```

`a === b` ?

Equal or not equal

```
const a = {};
const b = {};
```

`a === b` ?

NO

Equal or not equal

```
const a = { foo: 1 };
const b = { ...a };
```

`a === b` ?

Equal or not equal

```
const a = { foo: 1 };
const b = { ...a };
```

`a === b` ?

NO

Equal or not equal

```
const a = { foo: 1 };
const b = a;
b.foo = 2;
```

`a === b` ?

Equal or not equal

```
const a = { foo: 1 };
const b = a;
b.foo = 2;
```

`a === b` ?

YES

Equal or not equal

```
const a = [1, 2, 3];
const b = [1, 2, 3];
```

`a === b` ?

Equal or not equal

```
const a = [1, 2, 3];
const b = [1, 2, 3];
```

`a === b` ?

NO

Equal or not equal

```
const a = [];
const b = [];
```

`a === b` ?

Equal or not equal

```
const a = [];
const b = [];
```

`a === b` ?

NO

And the winner is… 📯

🔥 BONUS ROUND 🔥

🔥 BONUS ROUND 🔥

Equal OR not equal OR exception

```
const a = 0 / 0;
```

`a === a ?`

🔥 BONUS ROUND 🔥

Equal OR not equal OR exception

```
const a = 0 / 0;
```

`a === a` ?

🥁🥁🥁

🔥 BONUS ROUND 🔥

Equal OR not equal OR exception

```
const a = 0 / 0;
```

`a === a` ?

🥁🥁🥁

❌ NOT EQUAL ❌

🔥 BONUS ROUND 🔥

Equal OR not equal OR exception

```
const a = 0 / 0;
```

`a === a ?`

🥁🥁🥁

❌ NOT EQUAL ❌

`NaN !== NaN`

A MISSING RENDER

What’s broken and how do we fix it?

A MISSING RENDER – CAUSE

```
const v1 = {};
v1.foo = 1;

const v2 = v1;
v2.foo = 2;
```

`v1 === v2`

If this object is passed as a prop, the change of property `foo` will *not* trigger a re-render.

Shallow comparison will only compare the reference to the object, which is the same. Only a costly deep comparison would compare the contents.

A MISSING RENDER – OPTION 1

Property `foo` can be passed as prop directly.

That works well for simple cases: instead of passing one prop which is an object, instead spread the contents of the object.

BEFORE

Usage:

```
<MyComponent v={v} />
```

Component:

```
({ v }) => (<div>Hello {v.foo + v.bar}</div>)
```

AFTER

Usage:

```
<MyComponent foo={v.foo} bar={v.bar} />
```

```
<MyComponent {...v} />
```

Component:

```
({ foo, bar }) => (<div>Hello {foo + bar}</div>)
```

A MISSING RENDER – BAD LUCK

Option `1` is a nice idea, but it only solves the problem at the bottom of the tree. The same issue will arise at the level above!

Unless we pass `foo` from the highest component to the lowest, the problem is not solved.

A MISSING RENDER – OPTION 2

Creating a new object.

```
const v1 = { foo: 1, bar: 1 };
const v2 = { ...v1, foo: 2 };
```

A MISSING RENDER – BAD LUCK

Objects are often nested one in another, so in order to apply option 2, not only will the object have to be recreated, but its parent as well, and maybe even its parent’s parent, etc.

That’s still better than option 1, let’s do it.

THE REDUX WAY

Recreating the state tree to update one property

```
// in a reducer
switch (action.type) {
  case UPDATE_FOO:
    return { ...state, foo: action.foo };
  case UPDATE_BAR:
    return { ...state, bar: action.bar };
}
```

Reducers create a new object every time. That lets shallow comparisons detect a difference.

UPSIDE

It works!

Performance is improved by the use of `connect()` with specific selectors which allow components at the bottom of the render tree to react to changes without the help of their parents.

DOWNSIDE

Redux enforces a very specific event-driven architecture which can be quite heavy.

A MISSING RENDER – OPTION 3

If we have to change the way we perform updates, could we find a way to do it that does not require recreating a whole tree of objects? After all, we're trying to optimize performance so generating a lot of useless objects is not ideal.

Introducing… *Observables!*

OBSERVABLES

PROMISES

A Promise is something that will receive a value at some point in the future.

You register a callback to be notified when that value arrives.

Promise API:

- `then` *if the value arrives*
- `catch` *if there’s a fatal error*

OBSERVABLES

Observables are similar to Promises:

You register a callback to be notified when a new value arrives, but the number of values can go from 0 to infinity (you don’t know in advance).

Observable API:

- `subscribe` *to receive events*
    - `next` *if a new value arrives*
    - `error` *if there’s a fatal error*
    - `complete` *if it’s the end*

EXAMPLES

- A clock that ticks every second
- A server that sends events or notifications
- An input field that the user can modify

THE RXJS WAY

RxJS is an implementation of the Observable API with many helper functions to combine and modify them in interesting ways.

See RxMarbles.com for a demo of the kinds of operations RxJS can do.

THE RXJS WAY

Instead of storing your state in regular variables, you would create observables and store the values by publishing them on the observable:

```
const observable = new Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});
```

It’s like when you create a `Promise` and call `resolve(123)` to send the value.

THE RXJS WAY

There is a binding for React that allows you to receive the latest value from an Observable using a hook.

It is natively integrated in Angular and used for everything there.

THE RXJS WAY

UPSIDE

- No more objects to recreate.
- Advanced time-based functionality available when you need it.

DOWNSIDE

- Verbose and hard to decifer (low signal-to-noise).
- You have to replace all your values by observables.

THE MOBX WAY

What if you could have the best of both worlds?

- Use regular objects
- Be able to react to their changes.

This is what MobX offers.

THE MOBX WAY

Imagine your state is a simple counter. Now your counter is observable!

```
class Counter {
  count = 0;

  constructor() {
    makeAutoObservable(this); // enable MobX
  }

  increment() {
    ++this.count;
  }
}
```

THE MOBX WAY

MobX also takes care of subscribing your React components to the observables they use to react to changes automatically:

```
function MyComponent({ counter }) {
  return (
    <div>
      <span>You have clicked {counter.count} times.</span>
      <button onClick={counter.increment}>+1</button>
    </div>
  );
}

export default observer(MyComponent); // enable MobX
```

TAKEWAYS

Expressive immediate-mode code improves developer experience and maintainability but hurts performance.

Performance can be recovered but at a cost which is that regular JavaScript object changes are undetected if not recreated or observed.

Observables allow publishing and subscribing to changes but you have to instantiate and manage them individually.

RxJS allows manipulating observable data in many ways but is unwieldy for more common simple use cases.

MobX bridges the gap between standard objects, Observables, and React for maximum readability and ease of use while not covering rare advanced use cases.

Looking beyond the current state of JavaScript, there is a lot of thought being put into building *reactive systems* in the industry.

On the backend side everything related to *event-sourcing* and *CQRS* is very interesting.

There are also different approaches to this problem in the world of desktop GUIs with Qt’s QML which uses JavaScript but modified to be natively reactive.

THANK YOU FOR YOUR ATTENTION 🤗

Questions?