Evolution Engineering | ACCELERATING EVOLUTION

Typescript Bootcamp 2023

MobX 1

Kirill Voloshin
24 Feb 2023

Evolution Engineering | ACCELERATING EVOLUTION

What others are saying…

- I have built big apps with MobX already and comparing to the one before that was using Redux, it is simpler to read and much easier to reason about.

- The #mobx is the way I always want things to be! It’s really surprising simple and fast! Totally awesome! Don’t miss it!

source

The gist of MobX

MobX distinguishes between the following three concepts in your application:

- State
- Actions
- Derivations

mobx docs

Evolution Engineering | ACCELERATING EVOLUTION

MobX basics: observable state

State is the data that drives your application.

- state can be stored in any data structure you like: plain objects, arrays, classes, cyclic data structures etc (usually classes).
- just make sure that all properties you want to change (and react to) over time are marked as observable so MobX can track them.

```
import { makeAutoObservable } from "mobx";

class Circle {
  // observable (deep observability by default)
  radius = 0;

  constructor(){
    // making class instance observable
    makeAutoObservable(this);
  }
}
```

The basic idea behind observables is that they keep track of the derivations that they affect so that every time their value changes, they can update those derivations as well.

MobX basics: actions

An action is any piece of code that changes the state.

- you can group code that changes observables by marking it as an explicit action. That way MobX can automatically apply transactions for effortless optimal performance.

Explicit action – state mutation marked as action.

Implicit action – state mutation not marked as action.

Transaction – batching of synchronous code (state updates are grouped together, so there are no intermediate state values and corresponding reactions).

```
import { action, observable, makeObservable, configure } from
"mobx";

// since mobx strict mode warns when implicit non-wrapped actions
are used, we can turn off this warning by using
configure({ enforceActions: "never" });

class Circle {
   radius = 0;

   setRadius(newRadius: number){
      this.radius = newRadius;
   }

   constructor(initialRadius: number){
      this.radius = initialRadius;
      makeObservable(this, {
         radius: observable,
         setRadius: action.bound,
      });
   }
}

const circle = new Circle(1);
circle.setRadius(20); // explicit action
circle.radius = 10; // implicit action
```

MobX basics: derivations

Derivation is anything that can be derived from the state without any further interaction (automatically responds to state changes).

Examples of derivations:

- The user interface: **view = function(state)**
- Derived data, such as the number of remaining todos;
- Backend integrations, e.g., sending changes to the server;

MobX distinguishes between two kinds of derivations:

- Computed values – can always be derived from the current observable state using a pure function.
- Reactions – side effects happening when the state changes (bridge between imperative and reactive programming);

Evolution Engineering | ACCELERATING EVOLUTION

MobX basics: derivations / computed values

computed – value, which can be derived from observables.

- computed values are observable themselves -> can be used by other computed values and reactions;
- code inside computed is a tracking function itself - accessed observables (and other computeds) are notified that they are tracked.
- if tracked observable or computed changed, but the result of a corresponding computed did not, reactions depending on this computed are not retriggered.
- computed gets suspended if nothing tracks it.

```
import { makeAutoObservable } from "mobx";

class Circle {
  radius = 0;

   // computed
   get area(){
      // its tracking function
      return Math.PI * Math.pow(this.radius, 2);
   }

   constructor(initialRadius: number) {
      makeAutoObservable(this);
      this.radius = initialRadius;
   }
}
```

if you want to create a value based on the current state, use computed.
Computeds must be pure functions in terms of observables and other computeds.

Evolution Engineering | ACCELERATING EVOLUTION

MobX basics: derivations / reaction

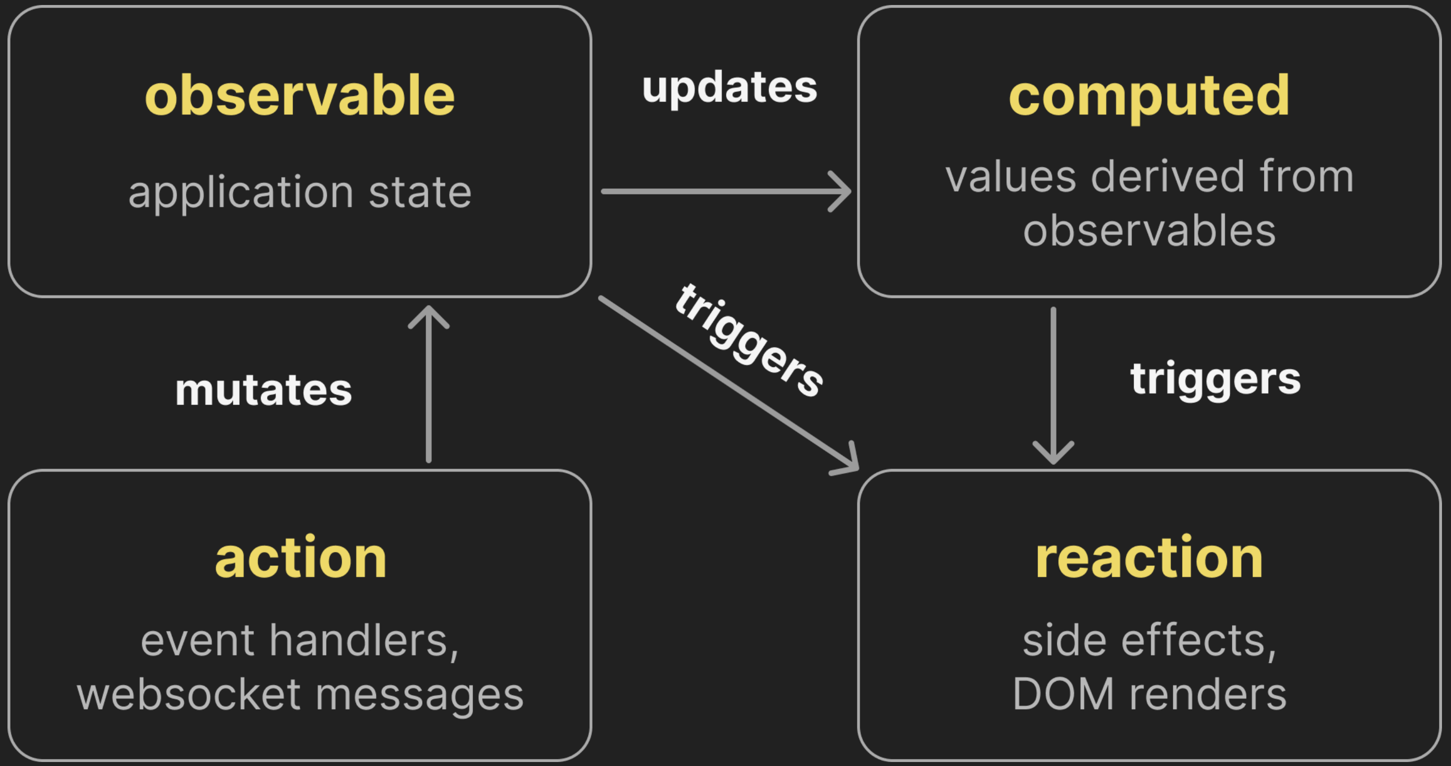
reaction - side effect caused by changes of the value dereferenced in its tracking function.

- reactions return a cleanup function, which can be called later to stop this reaction (unsubscribe from changes).

```
import { makeAutoObservable, reaction } from "mobx";

class BettingStore {
   betSpot = {
      bet: 9,
   }
   constructor(){
      makeAutoObservable(this);
   }
}
const bettingStore = new BettingStore();

const reactionDisposeFunction = reaction(
   // tracking function
   () => bettingStore.betSpot.bet,
   // side effect
   (newBet, oldBet, reactionItself) => {
      console.log(`bet changed from ${oldBet} to ${newBet}`);
   },
   // configuration object
   {
      name: "change of bet amount",
      fireImmediately: true, // run reaction right away (not waiting for changes of data in
the tracking function)
   }
);

// => "bet changed from undefined to 9" (fireImmediately)
bettingStore.betSpot.bet = 10;
// => "bet changed from 9 to 10"

reactionDisposeFunction(); // "deleting" this reaction.
bettingStore.betSpot.bet = 11;
// => no reaction, because reaction was disposed.
```

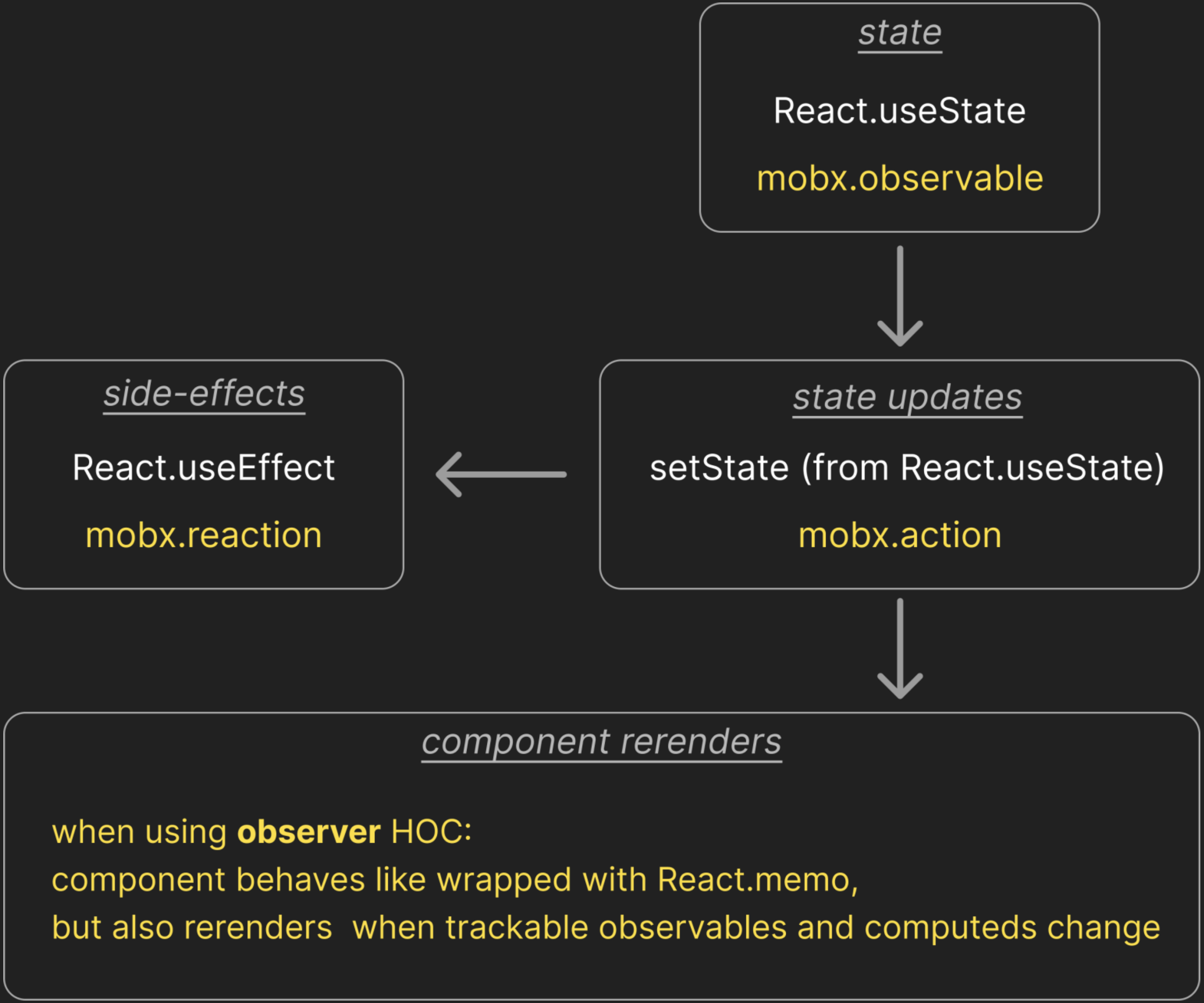
Evolution Engineering | ACCELERATING EVOLUTION

MobX overview


observable
application state
updates
computed
values derived from observables
triggers
mutates
triggers
action
event handlers, websocket messages
reaction
side effects, DOM renders

Evolution Engineering | ACCELERATING EVOLUTION

Analogy with react



when using **observer** HOC:
component behaves like wrapped with React.memo,
but also rerenders when trackable observables and computeds change

Evolution Engineering | ACCELERATING EVOLUTION

MobX integration with react

observer - HOC, which must wrap the component for it to rerender when accessed observable values are changed.

- observer HOC memoizes the component in a way that it will not be rerendered if observable values accessed inside it did not change (and, of course, if its props did not change - React.memo behavior).
- mobx store data is shared with react components using React Context.

render of this component can be derived from observables used inside it.

```
import { makeAutoObservable } from "mobx";
import { observer } from "mobx-react";
import * as React from "react";

class Counter {
  count = 0;
  increment(){
     this.count += 1;
  }
  constructor(){
    makeAutoObservable(this, {}, { autoBind: true });
  }
}
const CounterContext = React.createContext<Counter|null>(null);
const counter = new Counter();

const CounterComponent = observer(function(){
  const { increment } = React.useContext(CounterContext)!;
  const handleIncrement = React.useCallback(increment, [increment]);
  return (
    <>
      {counter.count}
      <button onClick={handleIncrement}> +1 </button>
    </>
  )
});

export default function App () {
  return (
    <CounterContext.Provider value={counter}>
      <CounterComponent />
    </ CounterContext.Provider>
  )
}
```

MobX-utils

MobX-utils provides an extensive series of additional utility functions, observables and common patterns for MobX.

- now() - functions, which returns the current date time as epoch number.

The function takes an interval as parameter, which indicates how often now() will return a new value.

If no interval is given, it will update each second.

If "frame" is specified, it will update each time a requestAnimationFrame is available.

```
import { autorun } from "mobx";
import { now } from "mobx-utils";

const start = Date.now();
autorun(() => {
   console.log("Seconds elapsed: ", Math.floor((now() - start) / 1000));
});

// => "Seconds elapsed: 0"
// => "Seconds elapsed: 1"
// => "Seconds elapsed: 2"
// => "Seconds elapsed: 3”
// ...
```

Useful materials:

- Mobx documentation: https://mobx.js.org/README.html
- Mobx source code: https://mobx.js.org/README.html
- Medium article - Becoming fully reactive: an in-depth explanation of MobX: https://medium.com/hackernoon/becoming-fully-reactive-an-in-depth-explanation-of-mobservable-55995262a254
- Tutorial using class components: https://egghead.io/courses/manage-complex-state-in-react-apps-with-mobx
- Book: https://iiunknown.gitbooks.io/mobxdocen/content/
- Several materials on mobx: https://github.com/mobxjs/awesome-mobx
- Understanding mobx (basic library implementation): https://github.com/jeromepl/understanding-mobx