



The first Hub for Developers  
Ztoupis Konstantinos

MobX

Code.Learn Program:  
React

# State management

- refers to the management of the state of one or more user interface controls
- the state of one UI control depends on the state of other UI controls

# local vs global scope



a bug can impact anything and everything in scope

huge global state objects → huge global bugs

tiny state objects → tiny localised bugs

# state and props

- both props and state are plain JS objects
- both props and state changes trigger a render update
- both props and state are deterministic. If your component generates different outputs for the same combination of props and state then you're doing something wrong

## state management primitives: immutable props and mutable state

# Good or Bad?

In React there is a lot of freedom to solving your structural and architectural problems

# Questions about store

- does this application need store?
- if so, how much state should store be responsible for?
- how smart should the smart components be?
- how dumb should the dumb components be?
- what components should be responsible for affecting and consuming global state?

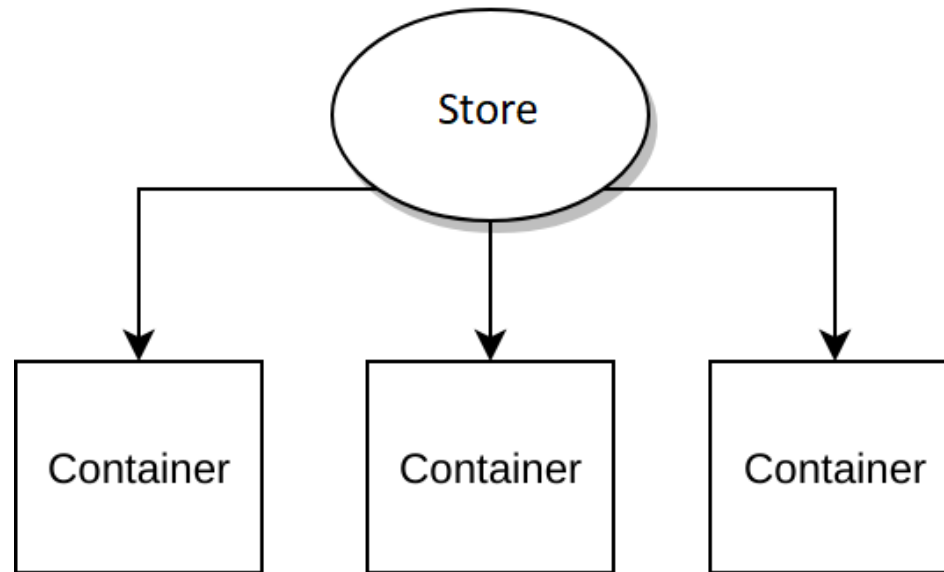
# State Definitions

- Global State
- Component State
- Relative State
- Provided State



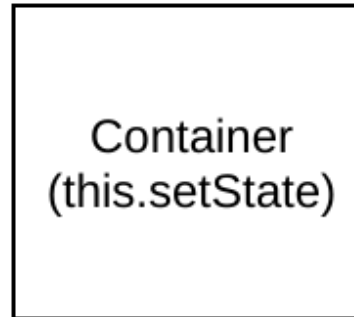
# Global State

State that is maintained outside of the component tree. The state that is held here is accessible from anywhere in the application.



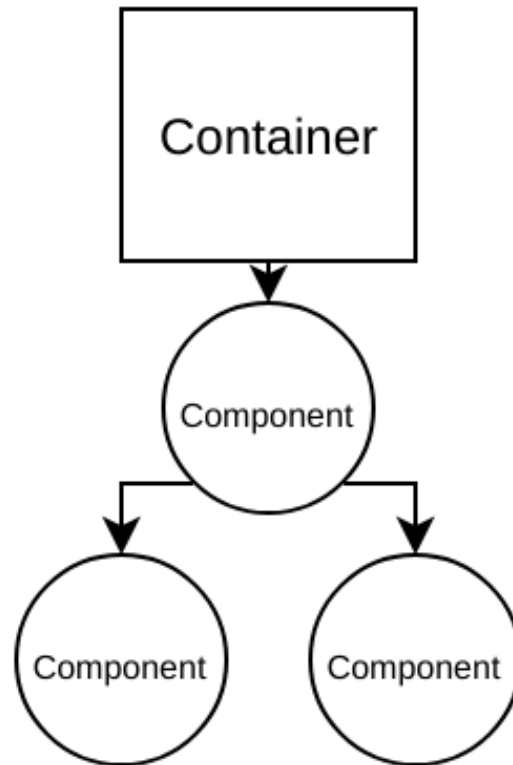
# Component State

State that is held within a component and manipulated with `this.setState`



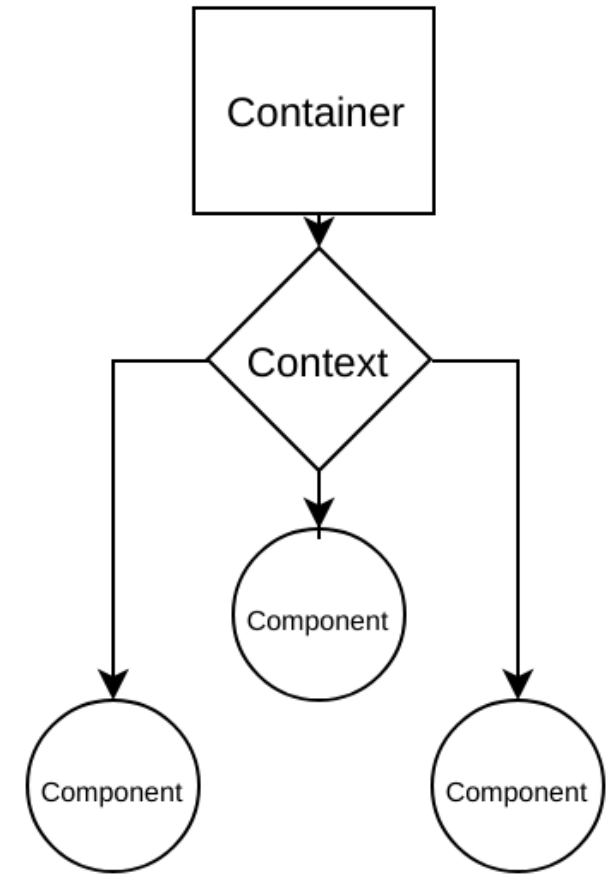
# Relative State

State that is passed from parent to child down the component tree.



# Provided State

The state is placed into a context by a provider. It can then be consumed by components individually without the need to be passed down the component tree.

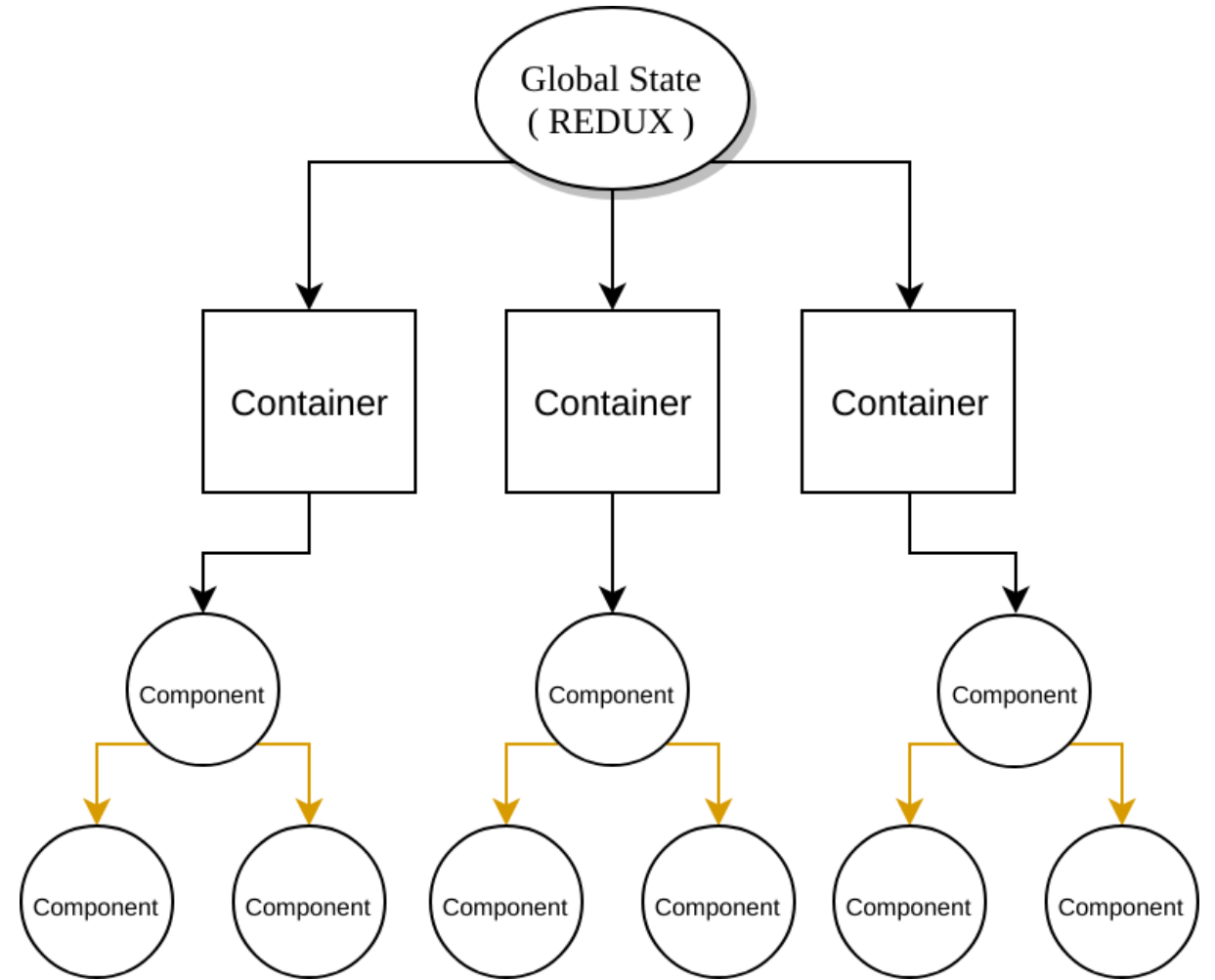


# Patterns

- Prop-drilling
- Redux-centric
- Provided State

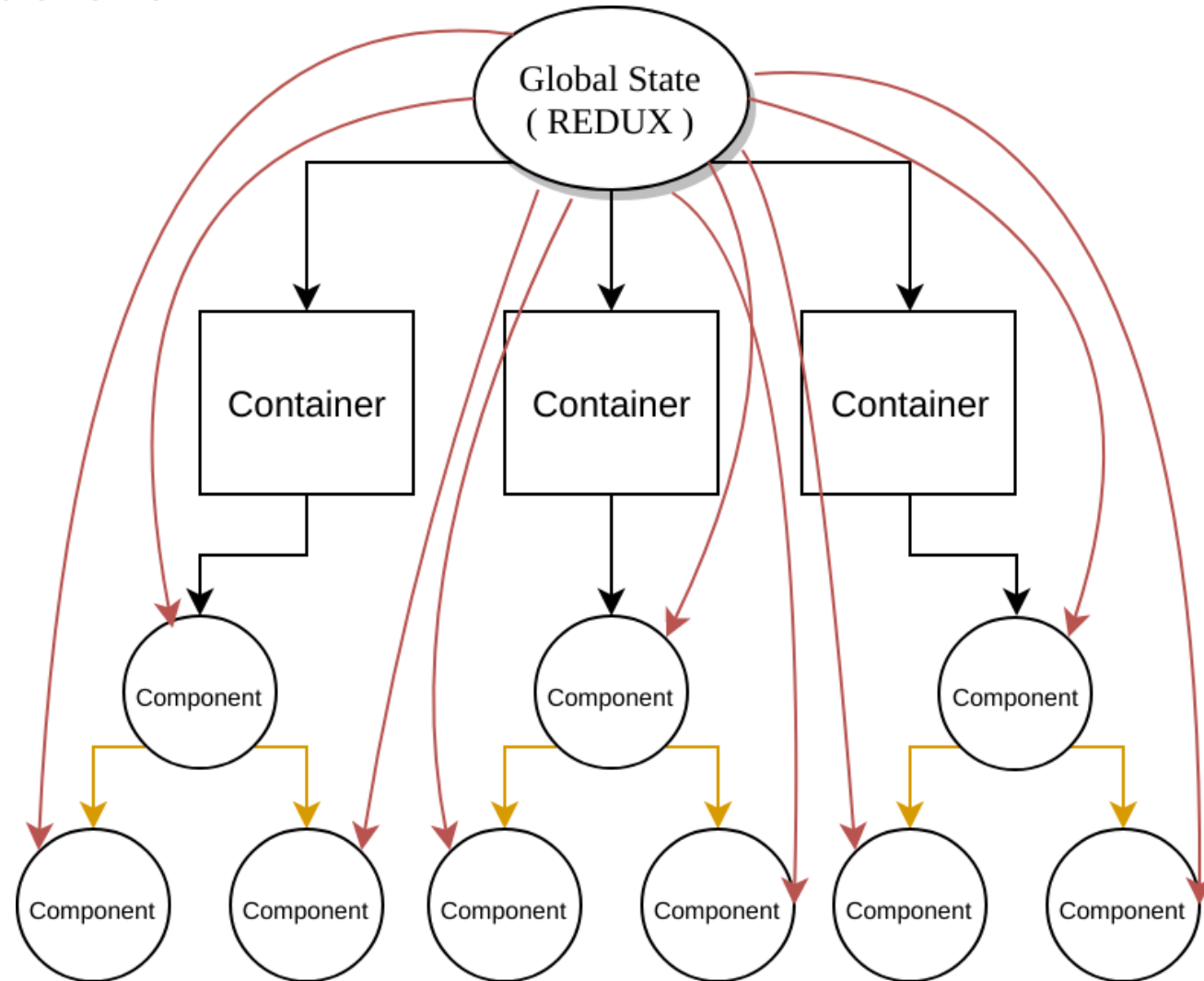
# Prop drilling

Three types of state: Global state, component state, relative state



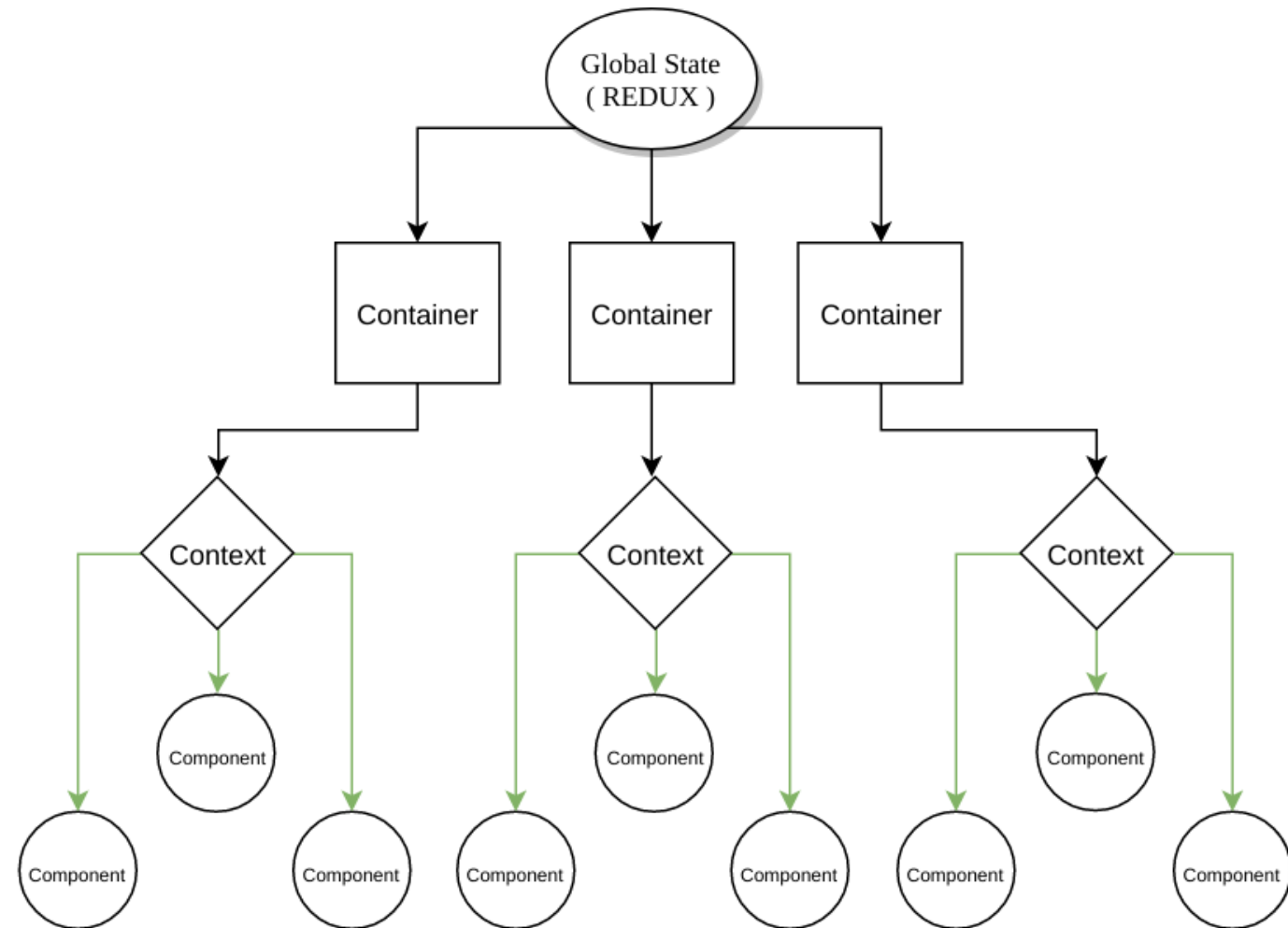
# Redux-centric

Components are a little bit smarter than before



# Provided State

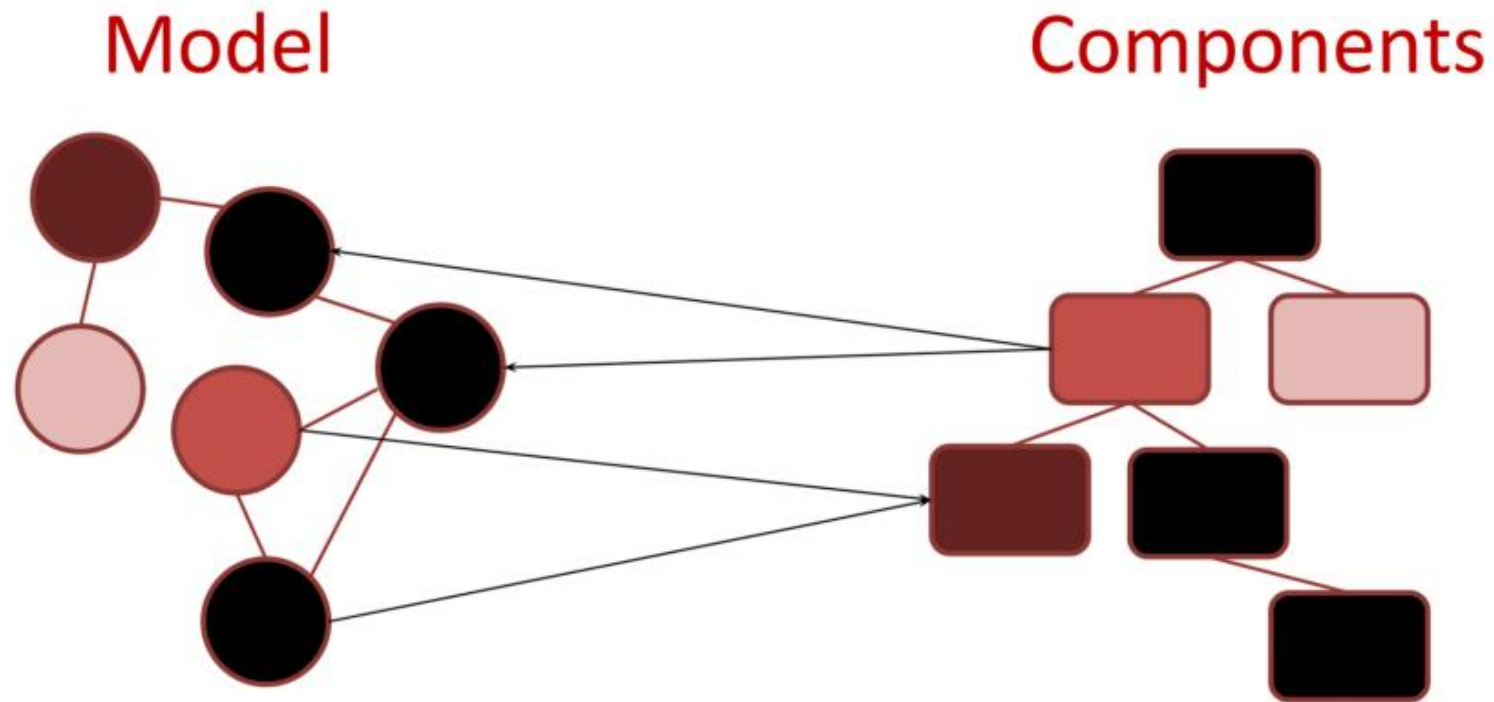
Containers manipulate and consume the global state. Components are only concerned with grabbing props from the context and displaying it.





# Who needs state management?

Building Frontends - Single page applications, is complex because there is so much state.



# Transforming state per component

general rule:

- a component *'should never transform data itself'*
- you *'should never write logic inside a UI component'*

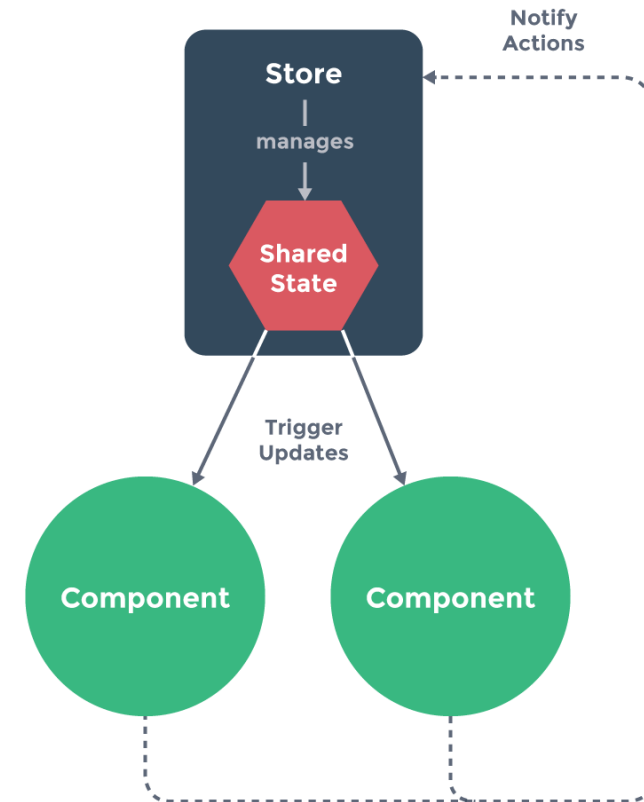
# Transforming state per component

Not transform data in a component:

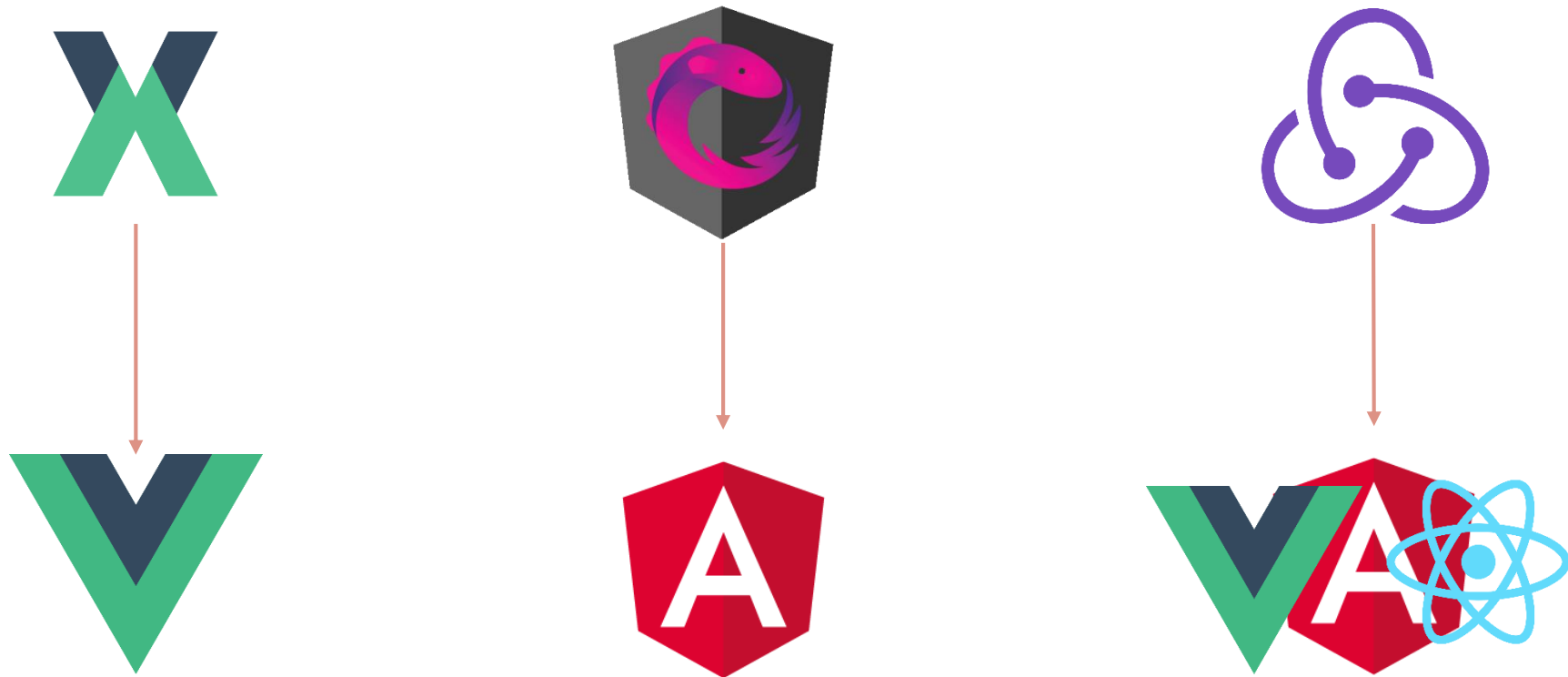
- coupling your component to a particular data structure means your component isn't reusable
- the transformation itself is single-use
- the transformation is harder to test inside a component
- it clutters components, making them far less readable

# State management

- state management: the toughest thing to get right
- unfortunately that's a lesson you learn over time, and without good planning you can lose your sanity



# State Tools



# react state management tools 1

- Controllerim: A state management library for React
- Dakpan: A tiny React state management library using the new React context.
- Freactal: Clean and robust state management for React and React-like libs.
- Freezer: A tree data structure that emits events on updates, even if the modification is triggered by one of the leaves, making it easier to think in a reactive way.
- Laco: Ultra lightweight state management for React and Inferno
- MobX: Simple, scalable state management
- react-contextual: is a tiny (less than 1KB) helper around React 16s new context api
- react-copy-write: Immutable state with a mutable API
- react-easy-state: minimal React state management with the power of ES6 Proxies
- redux + react-redux: Predictable state container for JavaScript apps +

# react state management tools 2

- Refunk: Simple React functional setState
- rosmaro + rosmaro-react: Visual automata-based programming for React
- Rematch: A Redux Framework
- Remx: Opinionated mobx
- Satcheljs: Satchel is a data store based on the Flux architecture. It is characterized by exposing an observable state that makes view updates painless and efficient.
- Stator: Simple, plain JavaScript state management with built-in support for React
- Sunfish: functional transaction based state management library
- tiny atom: Minimal, yet awesome, state management.
- Undux: Dead simple state management for React
- Unstated: State so simple, it goes without saying

# State is everywhere in JavaScript

- what the user sees
- what data are we fetching
- what URL are we showing to the user
- what items are selected inside the page
- the errors in the applications



# MboX

Simple, scalable state management, a standalone library



# Mobx

 Code.Hub

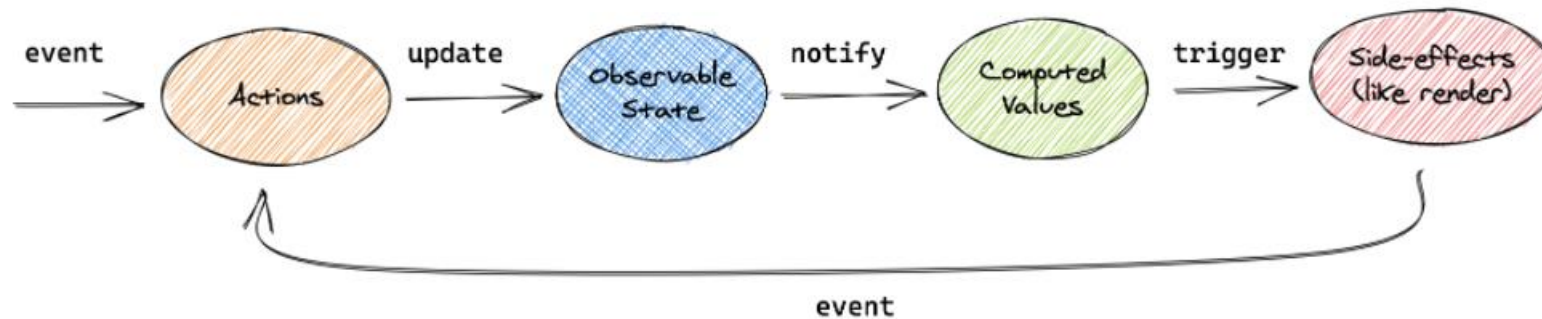
# Installation - Browser support

- `npm install mobx -save`
- `npm install mobx-react --save`

NPM Version	Support MobX version	Supported React versions	Supports hook based components
v7	6.*	16.8+	Yes
v6	4._ / 5._	16.8+	Yes
v5	4._ / 5._	0.13+	No, but it is possible to use <code>&lt;Observer&gt;</code> sections inside hook based components

# Introduction

Anything that can be derived from the application state, should be derived - Automatically



# The philosophy

MobX is a battle tested library that makes state management simple and scalable by transparently applying functional reactive programming (TFRP)

- Straightforward
- Effortless optimal rendering
- Architectural freedom

# Straightforward

No special tools are required, the reactivity system will detect all your changes and propagate them out to where they are being used.

# Effortless optimal rendering

All changes to and uses of your data are tracked at runtime, building a dependency tree that captures all relations between state and output.

# Architectural freedom

MobX is unopinionated and allows you to manage your application state outside of any UI framework. This makes your code decoupled, portable, and above all, easily testable.

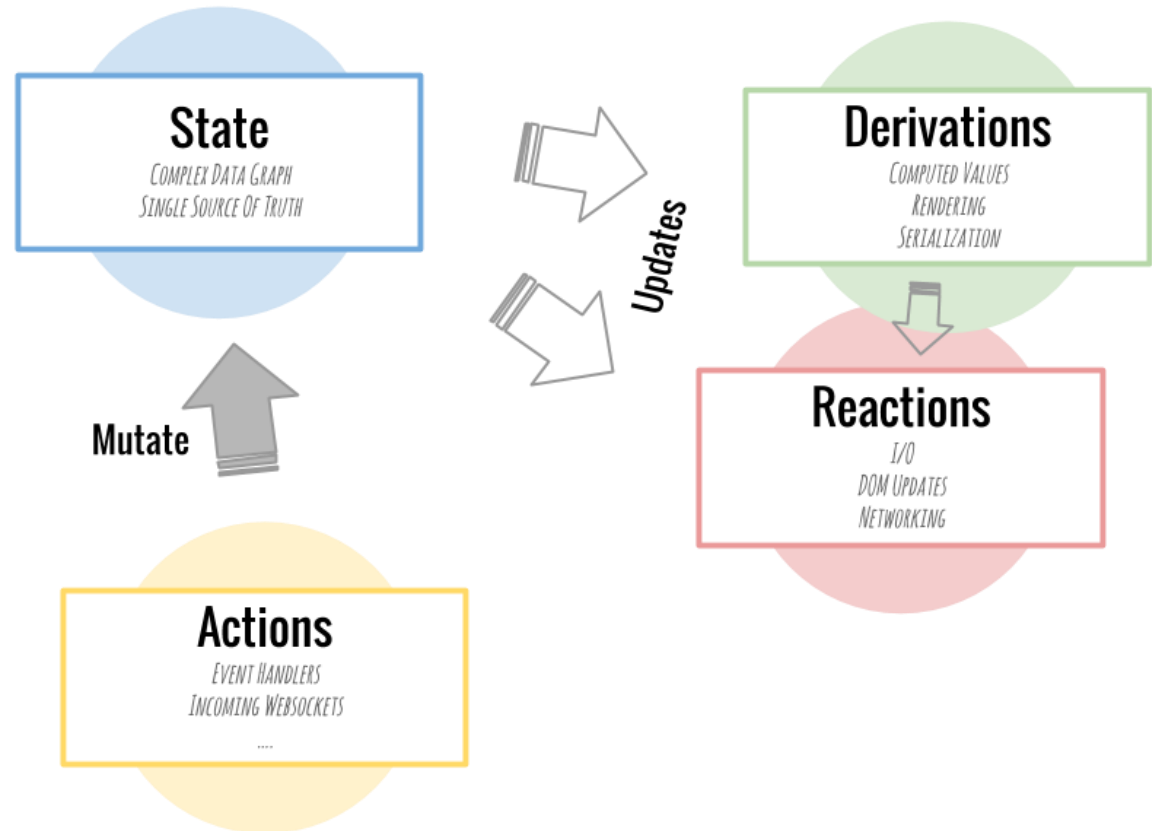
# Observe everything

- Any value can be observable
- Any component can be an observer
- Components (observers) will automatically re-render when observable values change



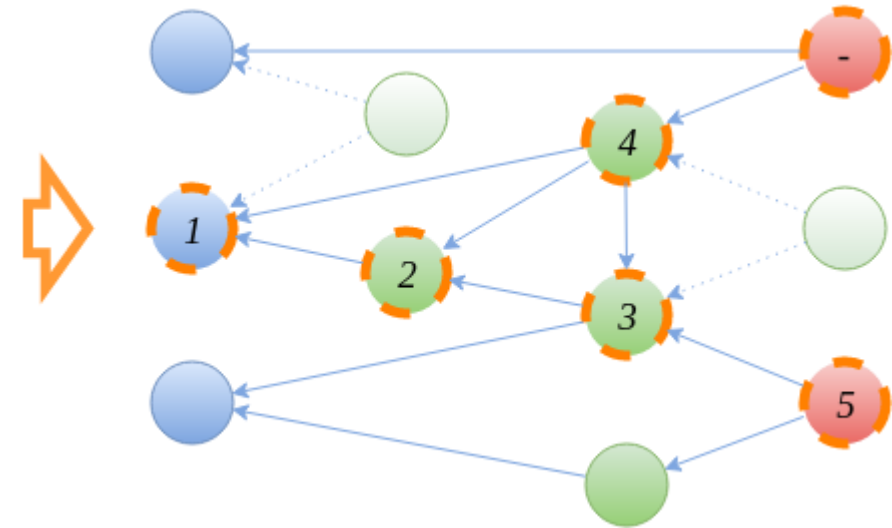
# Core concepts

- Observable state
- Computed
- Reactions
- Actions



# Virtual dependency state graph

- Observable state (blue)
- Computed values (green)
- Reactions (red)



# Observable state

```
import {makeObservable, observable,} from
"mobx"

class Doubler {
  value = 0;

  makeObservable(this, {
    value: observable
  })
}
```

# Computed values

```
import {makeObservable, observable, computed} from
"mobx"

class Doubler {
  value = 0;

  makeObservable(this, {
    value: observable,
    double: computed,
  })

  get double() {
    return this.value * 2
  }
}
```

Define values that will be derived automatically when relevant data is modified

# Reactions

- similar to a computed value
- produces a side effect incrementally updating the React component tree to patch the DOM, etc.

# Reactions

```
const App = () => {  
  const {increment} = store;  
  return (  
    <Counter  
      value={store.value}  
      increment={increment}  
    />  
  );  
};  
  
export default observer(App);
```

# Custom reactions

```
autorun (() => {  
  console.log(`the number of counter is {this.value}`);  
});
```

Custom reactions can simply be created using the `autorun`, `reaction` or `when` functions to fit your specific situations

# Actions

Somehow the state should be updated

```
import { ..., action } from
"mobx"

...
makeObservable(this, {
  value: observable,
  double: computed,
  increment: action
})

...
increment() {
  this.value++
}
```



# Provider and inject

Provider: component that can pass stores (or other stuff) using React's context mechanism to child components

inject: can be used to pick up those stores

```
import {Provider, inject} from 'mobx-react';  
  
<Provider store={store}>  
  <App />  
</Provider>  
  
...  
inject(['store'])(Counter)
```

# Asynchronous processes

```
import { when } from 'mobx';  
  
async() => {  
  await when(() => user.loggedOut)  
  alert("You have been successfully logged out!")  
}
```

# Observables to JavaScript

To convert a collection shallowly, the usual JavaScript mechanisms work:

```
const plainObject = { ...observableObject };  
const plainArray = observableArray.slice();  
const plainMap = new Map(observableMap);
```

# Simple and scalable

- one of the least obtrusive libraries you can use for state management
- not just simple, but very scalable
  - referential integrity is guaranteed
  - using classes and real references
  - simpler actions are easier to maintain
  - fine grained observability is efficient
  - easy interoperability

# Best Practices

- the stores represent the ui state
- separate your rest calls from the stores
- keep your business logic in stores
- don't create global store instances
- only the store is allowed to change its properties
- always annotate each component with observer
- use computed properties
- try to favor controlled components over uncontrolled components



MobX

vs



Redux

# Common



- are open-source libraries
- provide client-side state management
- are not tied to a specific framework
- have extensive support for React/React Native frameworks
- both Redux and MobX works well with react

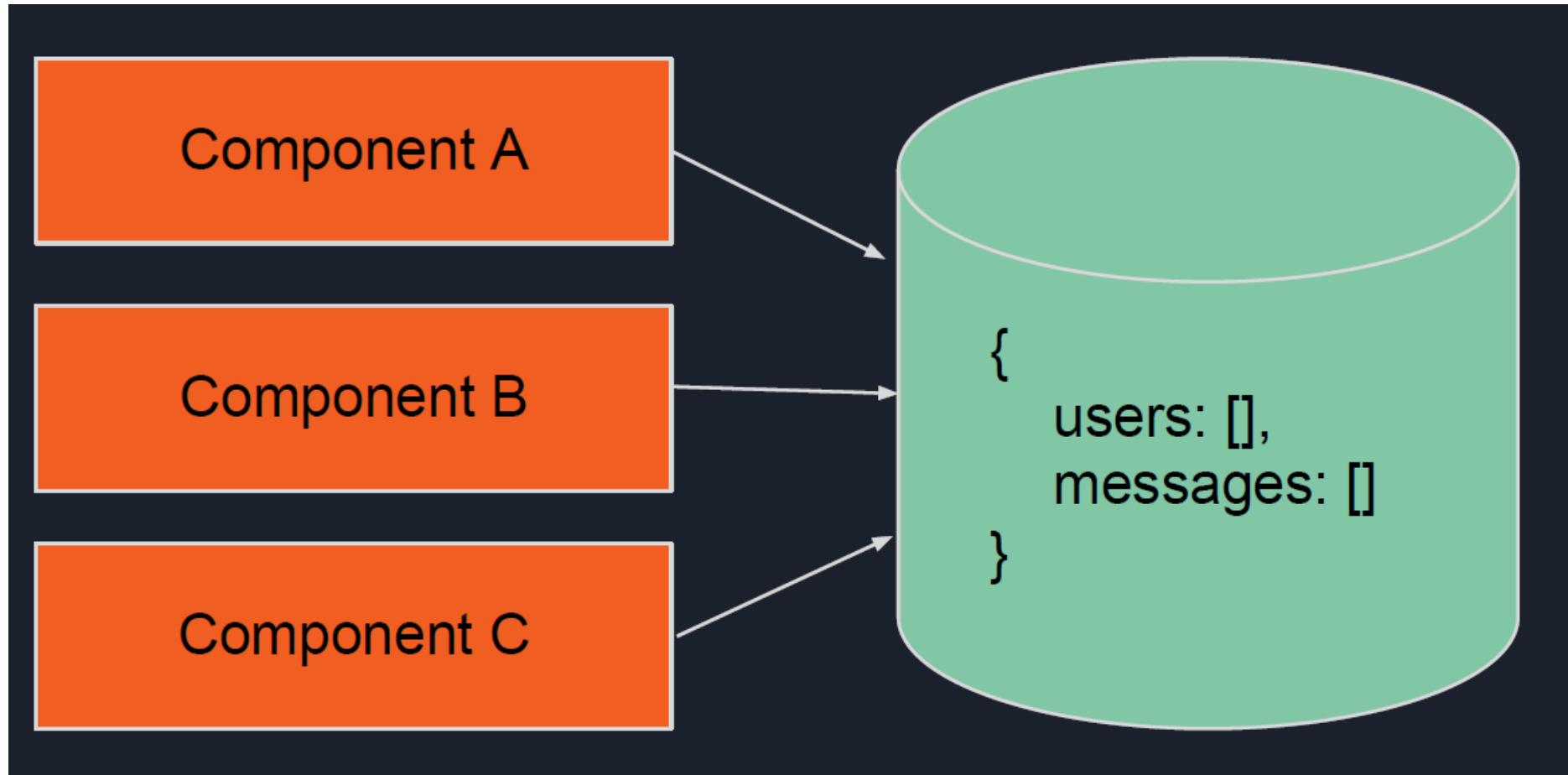
# Redux

- single store
- functional programming paradigm
- Immutable
- Pure
- explicit update logic
- plain JavaScript
- more boilerplate
- normalized state





# Single store

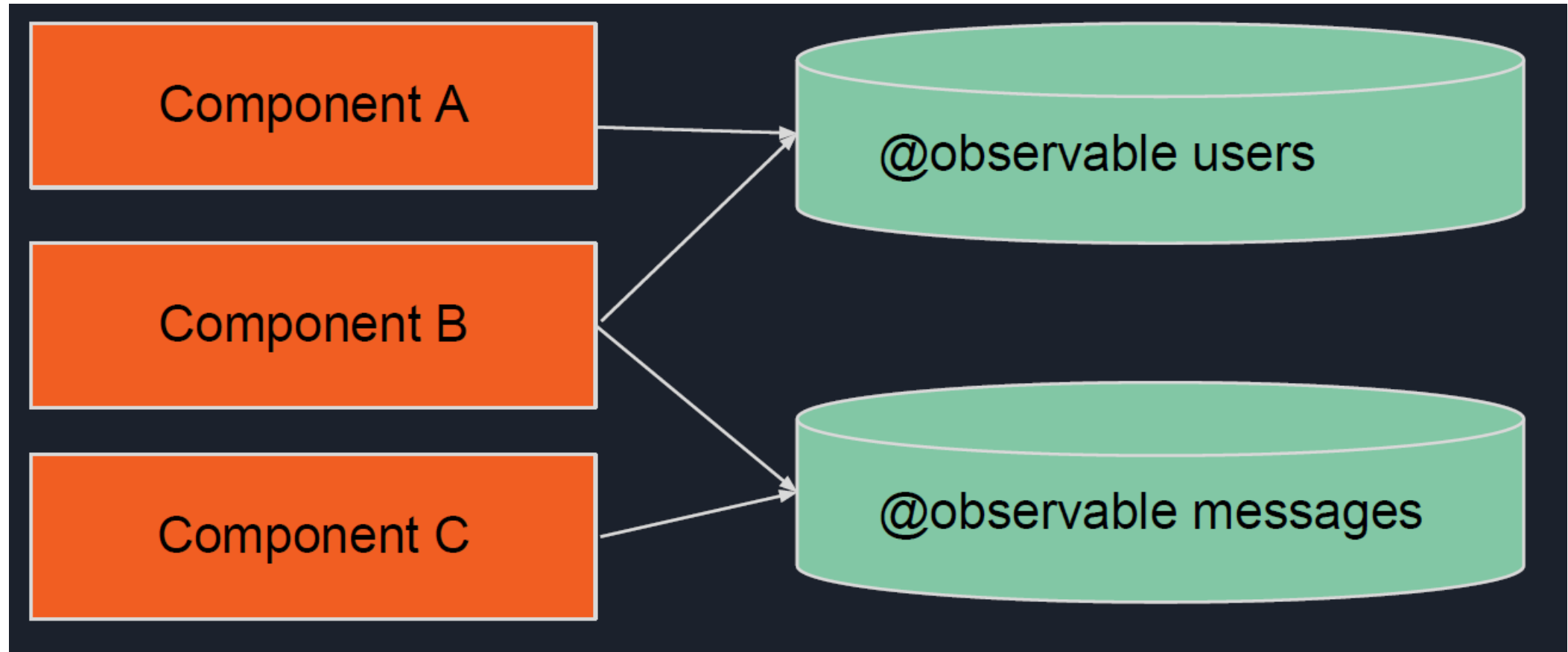


# MobX

- multiple stores
- object-oriented programming and reactive programming paradigms
- mutable
- impure
- implicit update logic
- “magic” JavaScript
- less boilerplate
- denormalized state
- nested state



# Multiple stores



# Redux vs MobX

- Single store vs Multiple Store
- Plain data vs Observable data
- Immutable vs mutable ( Pure vs Impure )
- Manual vs Automatic
- Normalized state vs Nested state
- Learning
- Testing
- Scale and Maintenance
- Developer Support



# Redux vs MobX

	useState	MobX	Redux
Location	Component	Component / Global	Global
Synchronicity	Asynchronous	Synchronous	Synchronous
Subscription	Implicit	Implicit	Explicit
Mutability	Mutable	Mutable	Immutable
Data structure	-	Graph	Tree
Observing Changes	-	Side Effect	Immutable -> append only

# Ask yourself

1. Is the application small and simple?
2. Prefer to build the app fast?
3. Large team looking for more maintainable code?
4. Complex app with scalable option?

