Reactive

- Virtual DOM Reconciliation
- State with hooks, context, and signals
- Todo example

Reactivity

- Reactivity can be broadly defined as the “automatic” update of the UI due to a change in the application's **state**
- As a developer, you can focus on the *state of the application* and let the framework *reflect that state in the user interface*

https://www.builder.io/blog/reactivity-across-frameworks

How Does Reactivity Actually Work?

- https://youtu.be/XB993rQ-5DY

Virtual DOM (VDOM) Reconciliation

- The VDOM is a lightweight representation of the UI in memory
- The VDOM is synchronized with the "real" DOM as follows:
  1. Save copy of current VDOM
  2. Components and/or application state updates the VDOM
  3. A re-render is triggered by framework
  4. Compare VDOM *before update* with VDOM *after update*
  5. **Reconcile the difference** by identifying a set of DOM patches
  6. Perform *patch operations* on real DOM
  7. Back to step 1

Node Difference Reconciliation Operations

- If the node type changes, the whole subtree is rebuilt

```
<div><p>Hello</p></div>
```

```
<section><p>Hello</p></section>
```

- If node type is the same, attributes are compared and updated

```
<div class="foo"><p>Hello</p></div>
```

```
<div class="foo bar"><p>Hello</p></div>
```

Sibling Difference Reconciliation Operations

- If a node is inserted into list of same node type siblings, all children would be updated (if more information isn't provided)

```
<ul>
  <li>Apple</li>
  <li>Banana</li>
</ul>
```

```
<ul>
  <li>Pear</li>
  <li>Apple</li>
  <li>Banana</li>
</ul>
```

first <li> must have changed from Apple to Pear,

second <li> must have changed from Banana to Apple,

etc.

Use keys for Better Child Reconciliation

- When updating children of same node type, use **key** prop
  - each key must be *stable* and *unique*
  - key should be assigned when data created, *not when rendered*

```
<ul>
  <li key="a">Apple</li>
  <li key="b">Banana</li>
</ul>
```

```
<ul>
  <li key="p">Pear</li>
  <li key="a">Apple</li>
  <li key="b">Banana</li>
</ul>
```

no other <li> with key-"p", so much be inserting a new child

Do not use the iteration index for key ids when rendering a list!

https://robinpokorny.medium.com/index-as-a-key-is-an-anti-pattern-e0349aece318

calendar.perfplanet.com

React’s diff algorithm

28th Dec 2013 by Christopher Chedeau

ABOUT THE AUTHOR

Christopher Chedeau (@vjeux) is a Facebook Software Engineer in the Photos Team

React is a JavaScript library for building user interfaces developed by Facebook. It has been designed from the ground up with performance in mind. In this article I will present how the diff algorithm and rendering work in React so you can optimize your own apps.

Diff Algorithm

Before we go into the implementation details it is important to get an overview of how React works.

```
var MyComponent = React.createClass({ render: function() { if (tl
```

At any point in time, you describe how you want your UI to look like. It is important to understand that the result of render is not an actual DOM node. Those are just lightweight JavaScript objects. We call them the virtual DOM.

React is going to use this representation to try to find the minimum number of steps to go from the previous render to the next. For example, if we mount `<MyComponent`

React’s diff algorithm (Dec 2013)

- https://calendar.perfplanet.com/2013/diff/

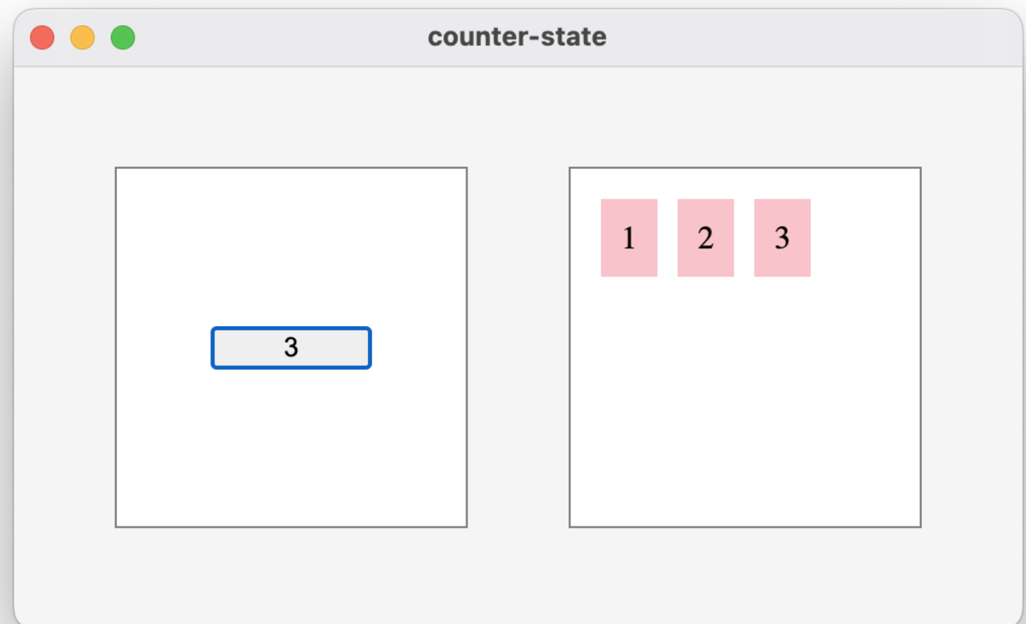
Approaches to Managing Application State

1. useState hook for local component state
   - pass state to children
2. useContext hook to access state
   - without passing as props
3. Signals

Redux is another popular method, but we won't cover that in CS 349

count-state

- Manage state with useState hooks
- State is stored in root component
  - useState hook
- Passed to children using props
  - Some "prop drilling"
- Demo



Hooks

Behind the scenes, hook functions like `setState` work by storing data in a sequence of "slots" associated with each component in the Virtual DOM tree. Calling a hook function uses up one slot, and increments an internal "slot number" counter so the next call uses the next slot. Preact resets this counter before invoking each component, so each hook call gets associated with the same slot when a component is rendered multiple times.

```
function User() {
  const [name, setName] = useState("Bob")    // slot 0
  const [age, setAge] = useState(42)         // slot 1
  const [online, setOnline] = useState(true) // slot 2
}
```

This is called call site ordering, and it's the reason why hooks must always be called in the same order within a component, and cannot be called conditionally or within loops.

counter-state (extra demo)

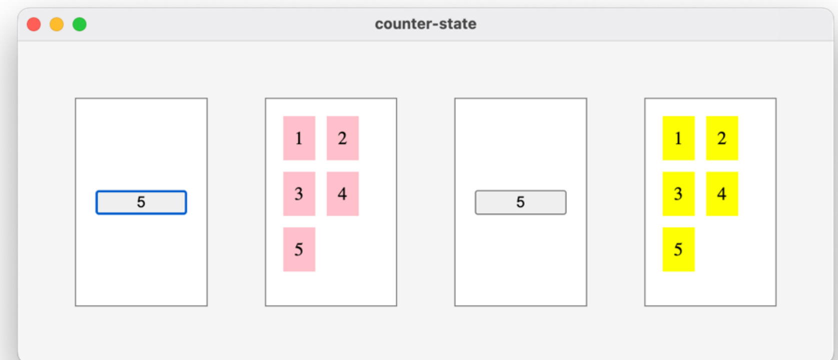
- Duplicate the Left and Right components
- Add another count state in "main.ts"

```
[count2, setCount2] = useState(0);
```

- Now use random hook call order:

```
let count, setCount, count2, setCount2;
if (Math.random() > 0.5) {
  [count, setCount] = useState(0);
  [count2, setCount2] = useState(0);
} else {
  [count2, setCount2] = useState(0);
  [count, setCount] = useState(0);
}
```

- Notice what happens ...



Hooks in Preact

- Functional methods to compose state and side effects
- useState
  - to get and set state
- useContext
  - access state context without prop drilling
- useRef
  - get a reference to a DOM node inside a functional component
- useEffect
  - trigger side-effects on state change
- useReducer
  - for complex state logic similar to Redux

(and several more …)

https://preactjs.com/guide/v10/hooks/

Custom Hooks

- Define a custom counter hook:

```
function useCounter() {
  const [value, setValue] = useState(0);
  const increment = useCallback(() => {
    setValue(value + 1);
  }, [value]);
  return { value, increment };
}
```

dependency argument

- Use the custom counter hook:

```
export default function App() {
  const { value, increment } = useCounter();
  return (
    <>
      <Left count={value} handleClick={increment} />
      <Right count={value} colour="pink" />
    </>
  );
}
```

Context

Pass state and other values to children without "prop drilling"

1. Define context object type

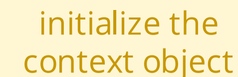
```
export type MyContextType = { colour: string; };
```

2. Create shared context object

```
export const MyContext = createContext({} as MyContextType);
```

3. Make context Provider node ancestor of components using context

```
<MyContext.Provider value={{colour: "lightgreen"}}>
   <MyComponent />
</MyContext.Provider>
```



4. Use context in child components

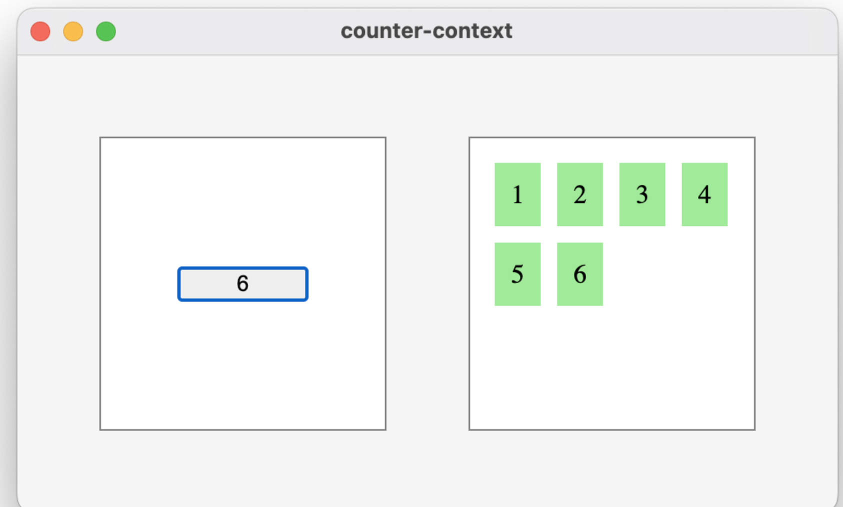
```
import { myContext } from "...";

function MyComponent() {
  const { colour } = useContext(MyContext);
  ...
```

https://preactjs.com/guide/v10/context

counter-context

- Manage state with useContext hooks
- CounterContext.ts
  - creates context object
- main.ts
  - uses context object with state hook
- LeftView and RightView components
  - no props, everything passed in context



Signals

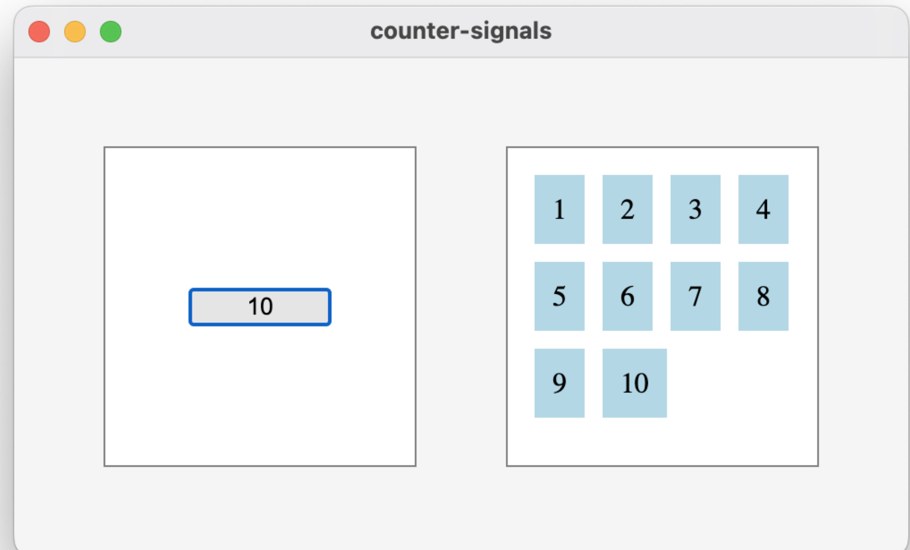
- State management module introduced by Preact
  - can be used with React and other frameworks too
- Not included by default, must install:

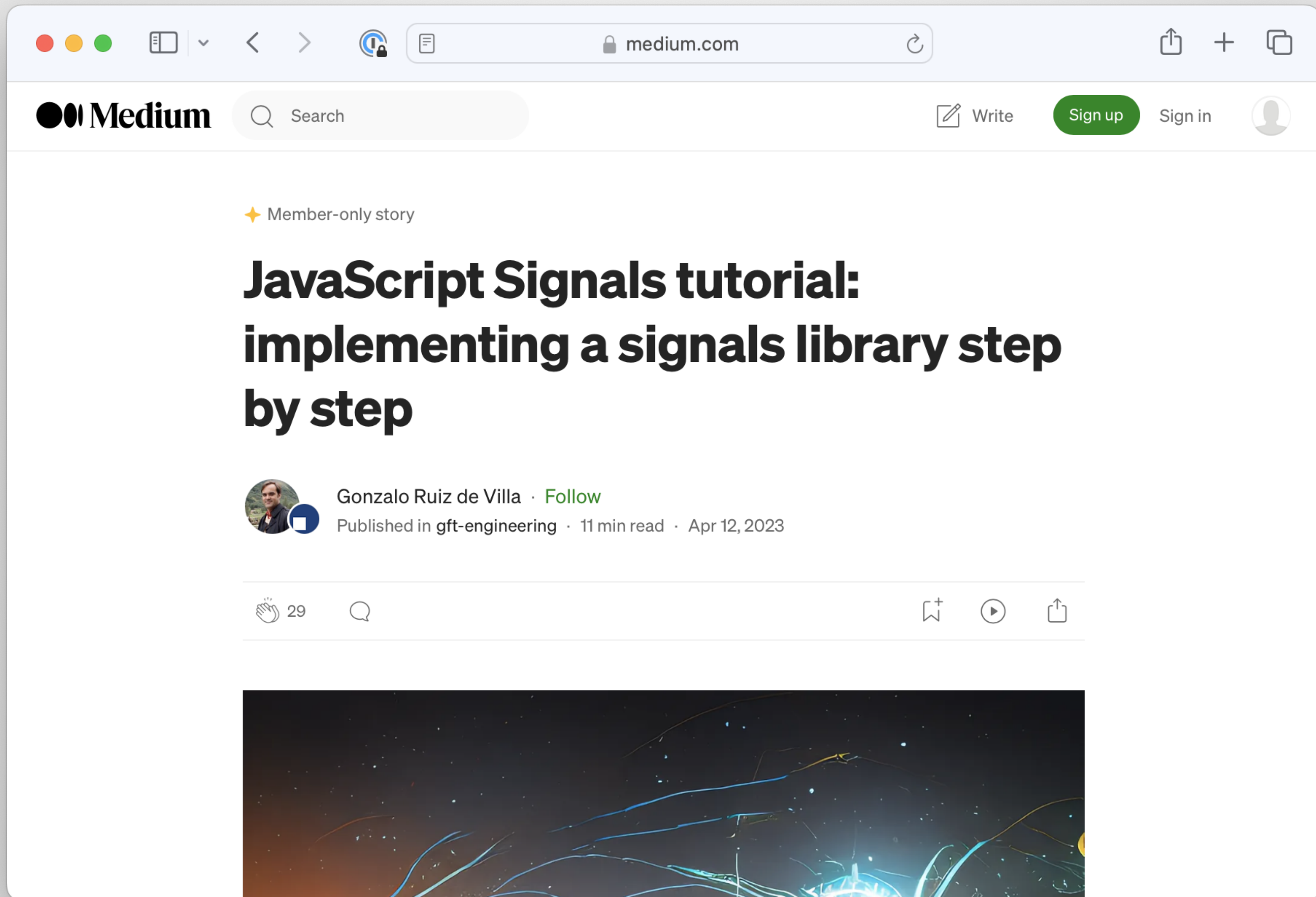
```
npm install @preact/signals
```

https://preactjs.com/blog/introducing-signals/

counter-signals

- Manage state with signals
- state.ts is like a Model
  - export signals with state, like `count`
  - can also export mutations, like `increment()`
- main.tsx
  - no state definition needed
- LeftView and RightView components
  - using State signals and mutation methods





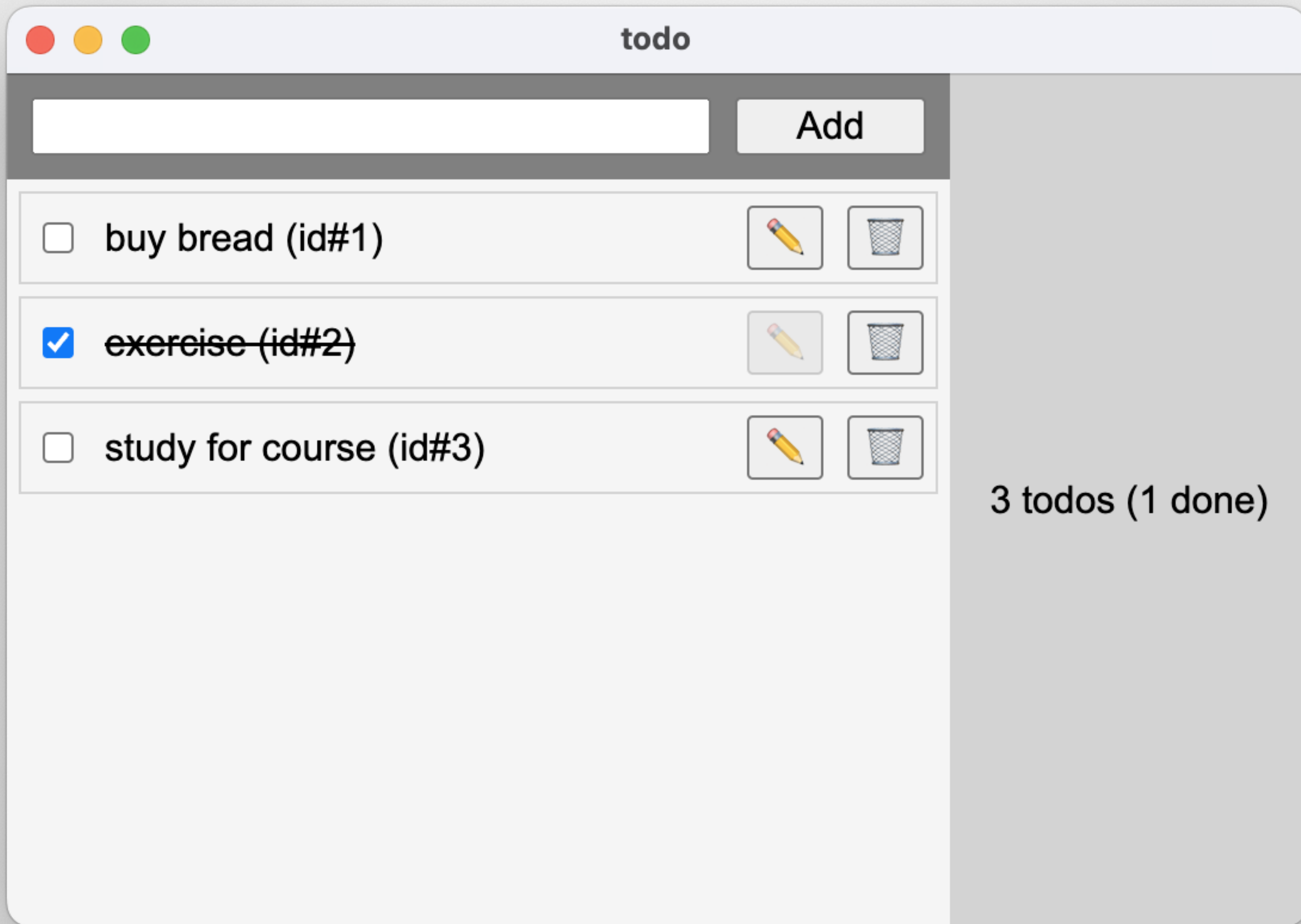
JavaScript Signals tutorial: implementing a signals library step by step (Medium member access)

- https://medium.com/gft-engineering/implementing-signals-in-javascript-step-by-step-9d0be46fb014

Suggestions for State with Signals

- Keep your state minimal and well-organized
  - each signal focused on a specific part of the state
- Use signals only where needed
  - too many signals can introduce performance issues
  - use signals to share state *between* different parts of application
  - for local app state, use useState() hook
- Keep components small and focused
  - “all-in-one” components handling several tasks makes state hard to manage
  - break UI down into small focused components that each manage/use a specific piece of state

https://polcode.com/resources/blog/signals-in-react-what-s-the-fuss-about/



todo demo using Preact and signals

todo

- A more complex app with array state
- state.ts has signals and mutations for app state
  - TS generics for signal typing
  - use of computed signal
  - mutations must *re-assign* the list signal value ref (e.g. arrays)
- List and TodoItem
  - map to create TodoItem children
  - TodoItem component uses todo prop (no state)
- Form
  - element ref with useRef