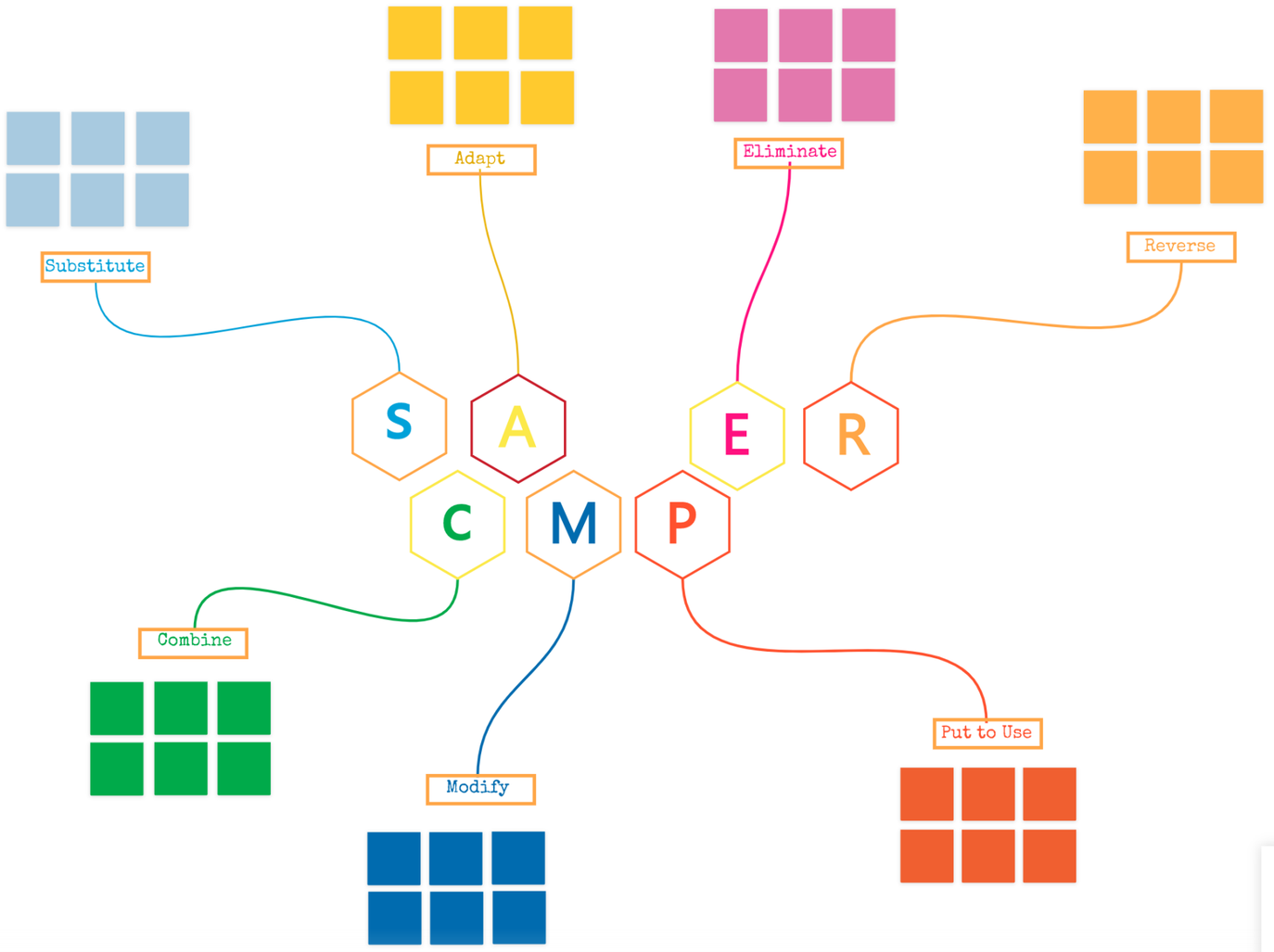
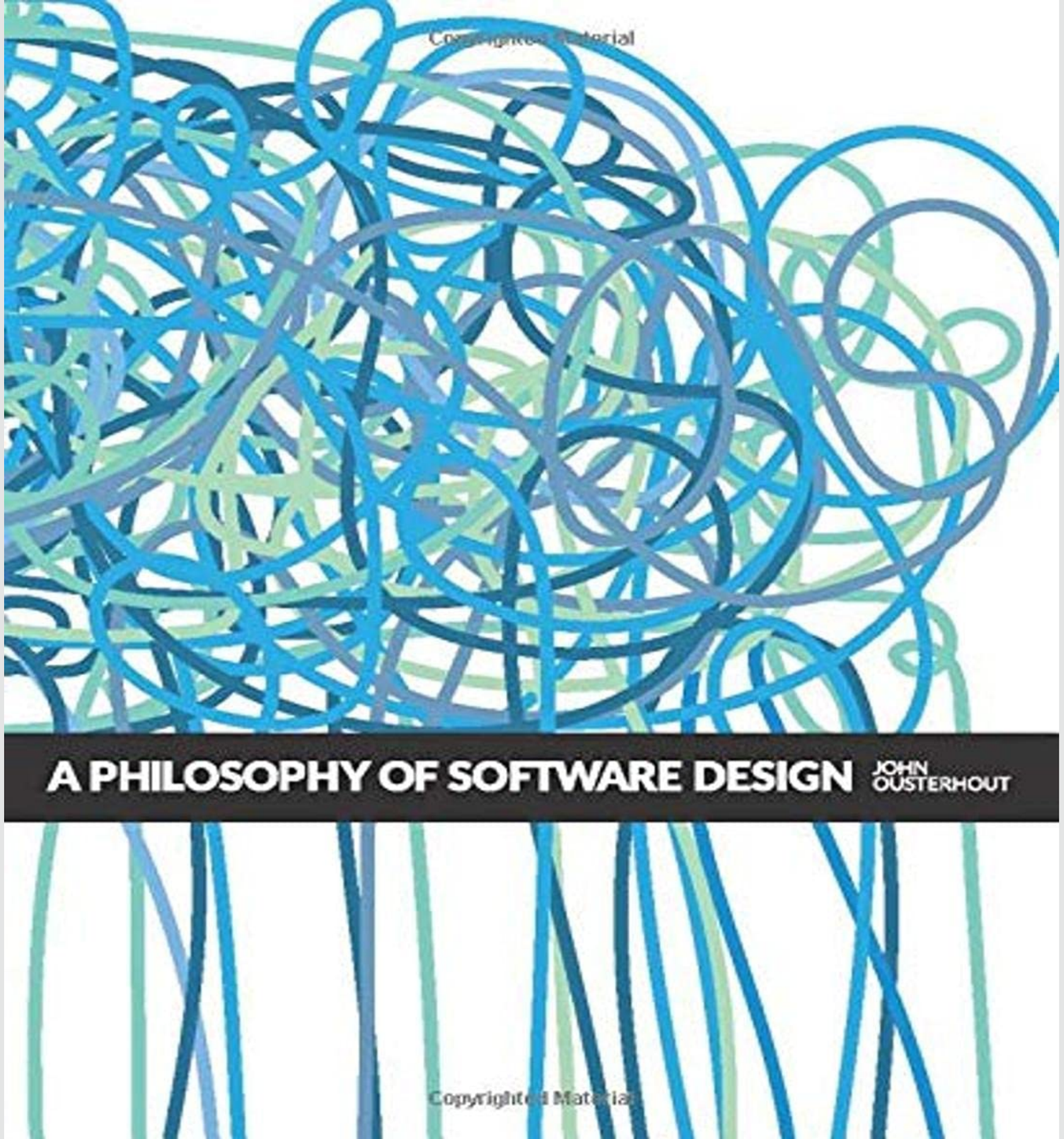
Principles and patterns in designing good React components

Collaboard

Great minds think together



Select
Note
Text
Doc
Image
Media
Shape
Line
Template
More


Copyrighted Material
A PHILOSOPHY OF SOFTWARE DESIGN
JOHN OUSTERHOUT
Copyrighted Material

Complexity

"Complexity is anything related to the structure of a software system that makes it **hard to understand and modify** the system."

Causes of complexity

— Obscurity

— Lack of good composition

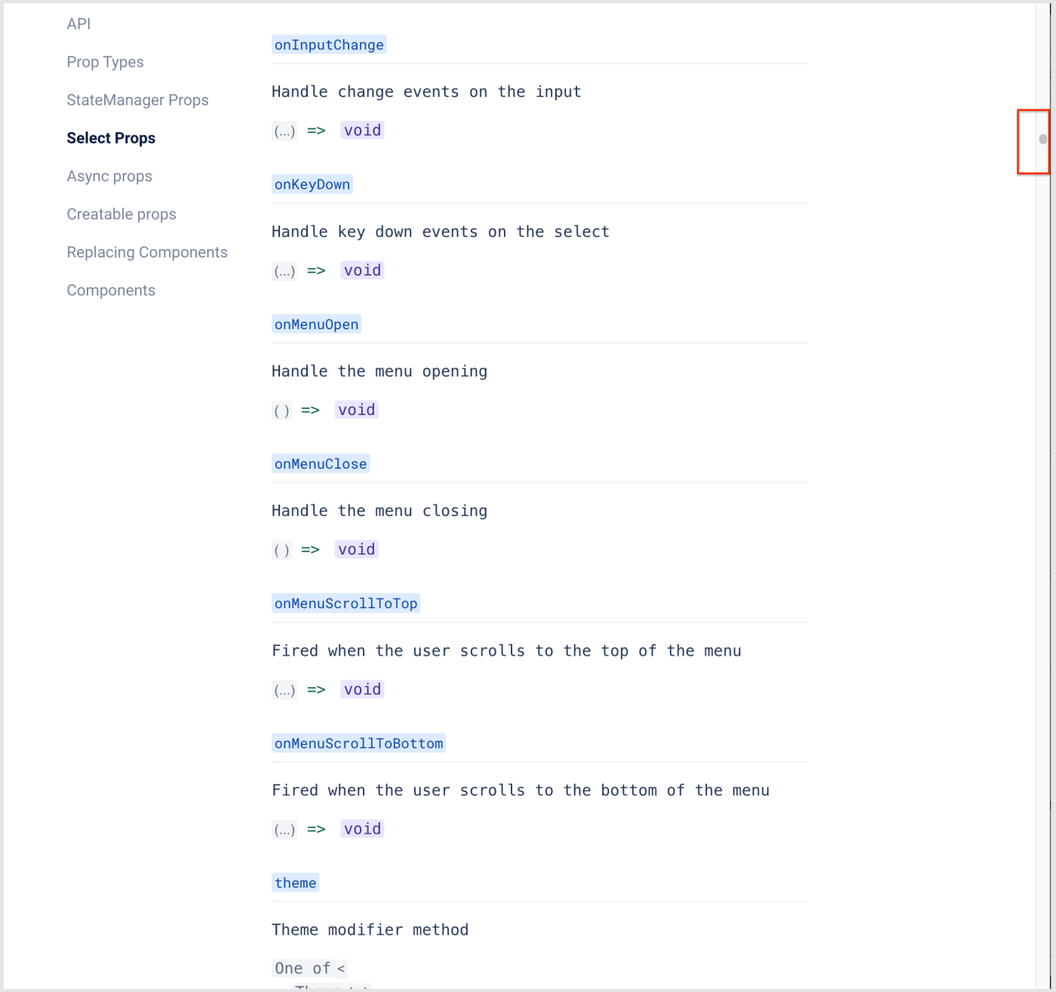
Obscurity

"Obscurity occurs when important information is not obvious."

Sometimes an implementation that requires more lines of code is actually simpler, because it reduces cognitive overload

Code is written once, read 100x more times.

Components with too many props



https://react-select.com/props

```
import Select from 'react-select'

const options = [
  { value: 'chocolate', label: 'Chocolate' },
  { value: 'strawberry', label: 'Strawberry' },
  { value: 'vanilla', label: 'Vanilla' }
]

const MyComponent = () => (
  <Select options={options} />
)
```

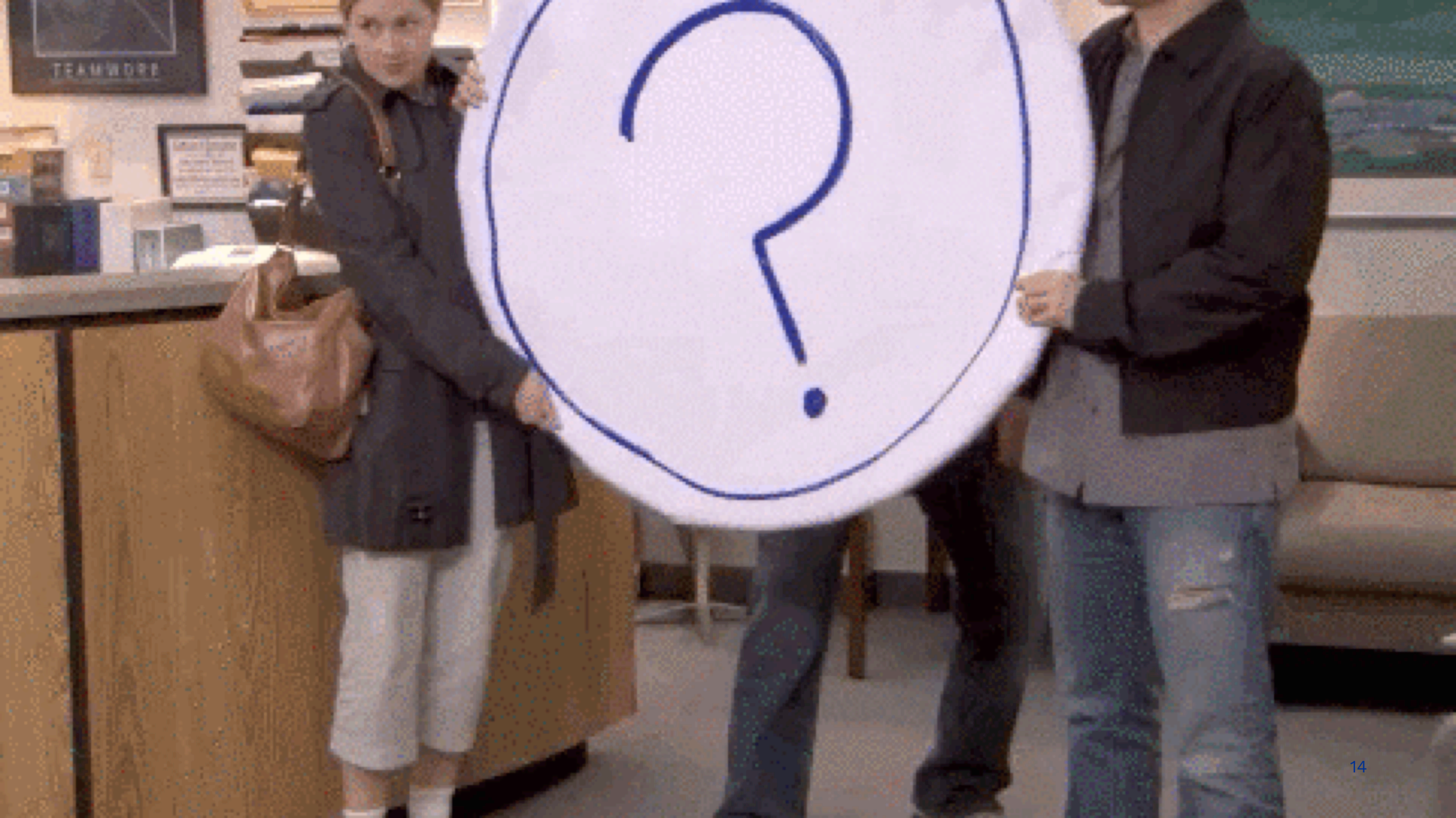
Reduce obscurity

Avoid optional parameters or props, prefer consistent explicit props

```
export type DialogState = {
  visible?: boolean;
  animated?: boolean | number;
};

// Used as hook
useDialog()

// Used as component
<Dialog></Dialog>
```

```ts
export type DialogState = {
  visible: boolean;
  animated: boolean | number;
};

// Used as hook
useDialog({ visible: true, animated: false })

// Used as component
<Dialog visible animated></Dialog>
```

- Make props or parameters more **discoverable**
- Make reading the code easier, with **less cognitive overload**
- Simplify implementation
  Less `props?.visible ?? false`


TEAMWORK

Reduce obscurity

Use **dimiscriminated union types** in TypeScript for less optional props and more explicit-consistent props

```ts
export type ShippingAddress = {
  name: string;
  street: string;
  city: string;
  postalCode: string;

  isPickup: boolean;
  pickupCompany?: string;
};

export type PaymentMethod = {
  paymentMethod: string;
  cardNumber: string;
  cardExpiration: string;
  cardCode: string;
};
```

```ts
export type ShippingAddress = {
  name: string;
  street: string;
  city: string;
  postalCode: string;

  isPickup: boolean;
  pickupCompany?: string;
};

export type PaymentMethod = {
  paymentMethod: string;
  cardNumber: string;
  cardExpiration: string;
  cardCode: string;
};
```

Discriminated union types - Avoid invalid states

```
type CommonInfo = {
  name: string;
  street: string;
  city: string;
  cap: string;
};

type SendToHome = CommonInfo & { kind: 'SendToHome' };

type SendToPickupPoint = CommonInfo & { kind: 'SendToPickupPoint'; company: string };

type ShippingAddress = SendToHome | SendToPickupPoint;
```

— `kind` is called discriminant property

Type narrowing again

```
function handleShipping(address: ShippingAddress) {
  // typeof address.kind === 'SendToHome' | 'SendToPickupPoint'
  console.log(address.company); // TS Error

  if (address.kind === 'SendToHome') {
    // typeof address.kind === 'SendToHome'
    console.log(address.company); // TS Error
    return;
  }

  // typeof address.kind === 'SendToPickupPoint'

  if (address.kind === 'SendToPickupPoint') {
    console.log(address.company); // Okay and with autocomplete
    return;
  }

  // typeof address.kind === never
  assertUnreachable(address.company)
}
```

Lack of composition

Composition is at the core of how we program software

Composition

```
<Modal
  isOpen={modalIsOpen}
  headerLabel="Modal header"
  contentLabel="Content label"
>
  <button onClick={closeModal}>close</button>
  <div>I am the modal content</div>
</Modal>
```

Functions are more composable than string or even objects

— Prefer function props

— Render props

— Prefer component composition than complex props

```
<Modal
  isOpen={modalIsOpen}
  header={() => <Text>Modal header</Text>}
  contentLabel={() => <Text bold>Modal header</Text>}
>
  <button onClick={closeModal}>close</button>
  <div>I am the modal content</div>
</Modal>
```

```
<Modal isOpen={modal}>
  <ModalHeader>Modal header</ModalHeader>
  <ModalBody>
    Modal content
  </ModalBody>
  <ModalFooter>
    <Button color="primary" onClick={toggle}>
      Do Something
    </Button>
    <Button color="secondary" onClick={toggle}>
      Cancel
    </Button>
  </ModalFooter>
</Modal>
```

Reakit

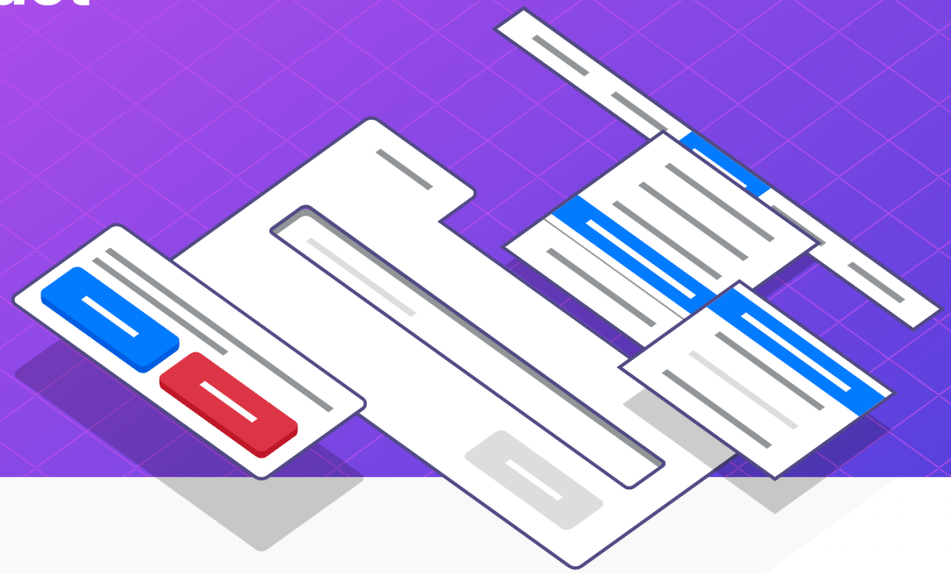
Start typing to search

DOCUMENTATION NEWSLETTER GITHUB

Build accessible rich web apps with React

GET STARTED COMPONENTS

VIEW REAKIT ON GITHUB ★ 6270



Accessible

Reakit strictly follows **WAI-ARIA 1.1** standards. All components come with proper attributes and keyboard interactions out of the box. Learn more

Composable

Reakit is built with composition in mind. You can leverage any component or hook to create new things. Learn more

Customizable

Reakit components are unstyled by default in the core library. Each component returns a single HTML element that accepts all HTML props, including `className` and `style`. Learn more

Tiny & Fast

Reakit components are built with modern React and follow best practices. Each imported component will add from a few bytes to up to `3 kB` into your bundle. Learn more

Ariakit

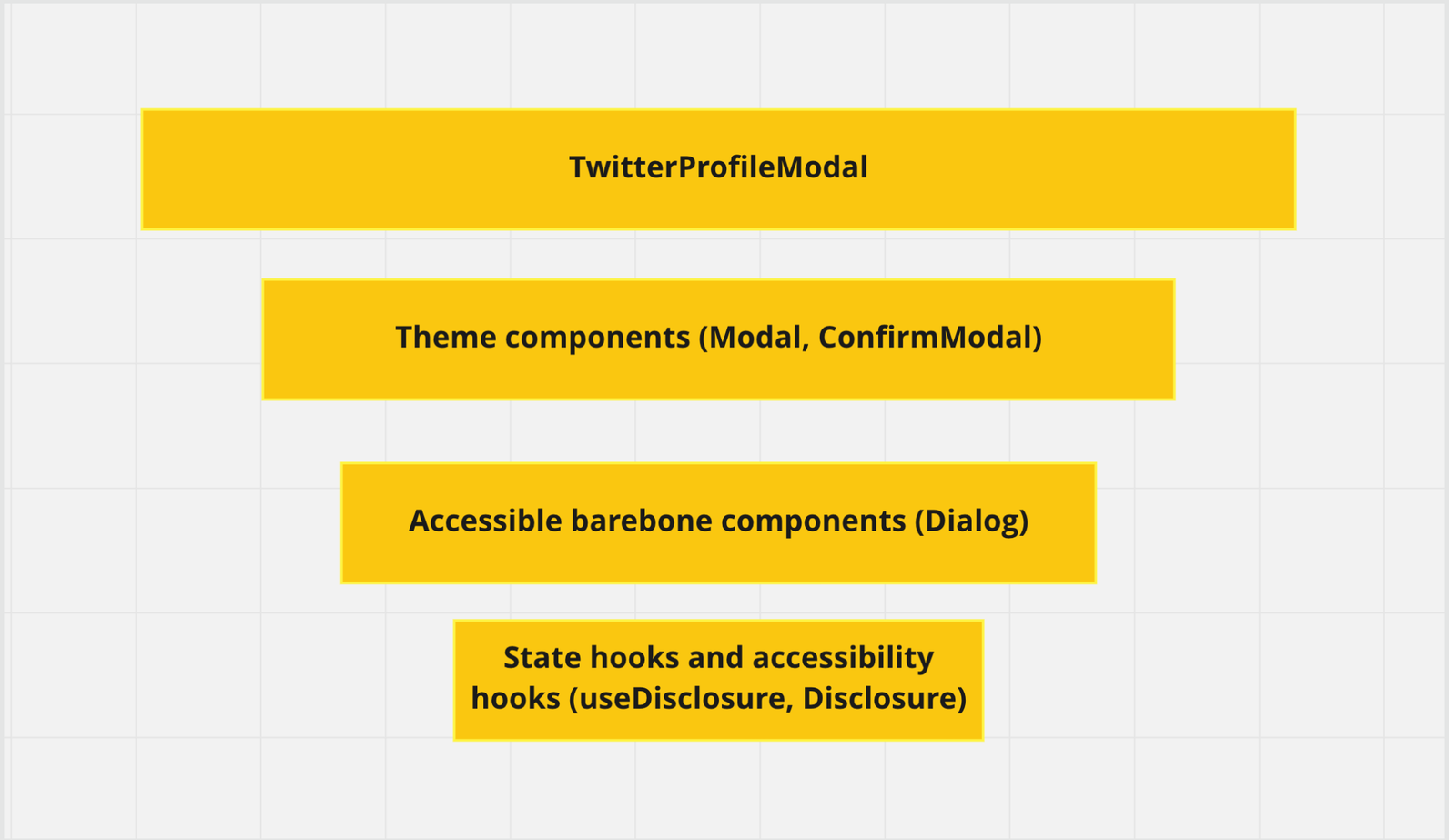
```
import { useDialogState, Dialog, DialogDisclosure } from "reakit/Dialog";

function Example() {
  const dialog = useDialogState();
  return (
    <>
      <DialogDisclosure {...dialog}>Open dialog</DialogDisclosure>
      <Dialog {...dialog} aria-label="Welcome">
        Welcome to Verona!
      </Dialog>
    </>
  );
}
```

```
function useDelayedDialogState({ delay, ...initialState }) {
  const disclosure = useDialogState(initialState);
  const [transitioning, setTransitioning] = React.useState(false);

  return {
    ...disclosure,
    transitioning,
    toggle: () => {
      setTransitioning(true);
      setTimeout(() => {
        disclosure.toggle();
        setTransitioning(false);
      }, delay);
    },
  };
}
```

```
function Example() {
  const dialog = useDelayedDialogState({ delay: 200 });
  return (
    <>
      <DialogDisclosure {...dialog}>Open dialog</DialogDisclosure>
      <Dialog {...dialog} aria-label="Welcome">
        Welcome to Verona!
      </Dialog>
    </>
  );
}
```

```
// "as" prop
<Disclosure as={Button}>Show</Disclosure>

// render prop
<Disclosure>
  {(props) => <Button type="button {...props}>Show</Button>}
</Disclosure>
```

Software is built on-top-of software (i.e. composed)



What is an Effect really?

— Variable mutations
— A DOM mutation or listening
— Canvas mutations
— Data fetching and revalidation
— WebSocket and WebRTC messages

```
useEffect(() => {
  if (!options.animated) {
    return undefined;
  }

  const raf = window.requestAnimationFrame(() => {
    if (options.visible) {
      setTransition('enter');
    } else if (animating) {
      setTransition('leave');
    } else {
      setTransition(null);
    }
  });

  return () => window.cancelAnimationFrame(raf);

}, [options.animated, options.visible, animating]);
```

API Effects

```
function getUserProfile(userId: string): Promise<UserProfile>

const [profile, setProfile] = useState<UserProfile | null>(null)

useEffect(() => {
  getUserProfile(userId).then(profile => setProfile(profile))
}, [userId]);
```

API Effects

```
type RequestState =
| { isLoading: false, profile: undefined, error: null }
| { isLoading: true, profile: undefined, error: null }
| { isLoading: false, profile: UserProfile, error: null }
| { isLoading: false, profile: undefined, error: Error }

const [requestState, setRequestState] = useState<UserProfile>(
  { isLoading: false, profile: undefined, error: null }
)

useEffect(() => {
  setRequestState({ isLoading: true, profile: undefined, error: null })

  getUserProfile(userId).then(profile => setRequestState(
    { isLoading: false, profile, error: null }
  )).catch(error => setRequestState({ isLoading: false, profile: undefined, error }))
}, [userId]);
```

React Query

```
const { isLoading, data: profile, error } = useQuery(
  ["userProfile", userId],
  () => getUserProfile(userId)
)
```

Real world data fetching features [1]

- Caching
- Retries on failure
- Deduping multiple requests for the same data into a single request
- Updating "out of date" data in the background
- Knowing when data is "out of date"
- Performance optimizations like pagination and lazy loading data

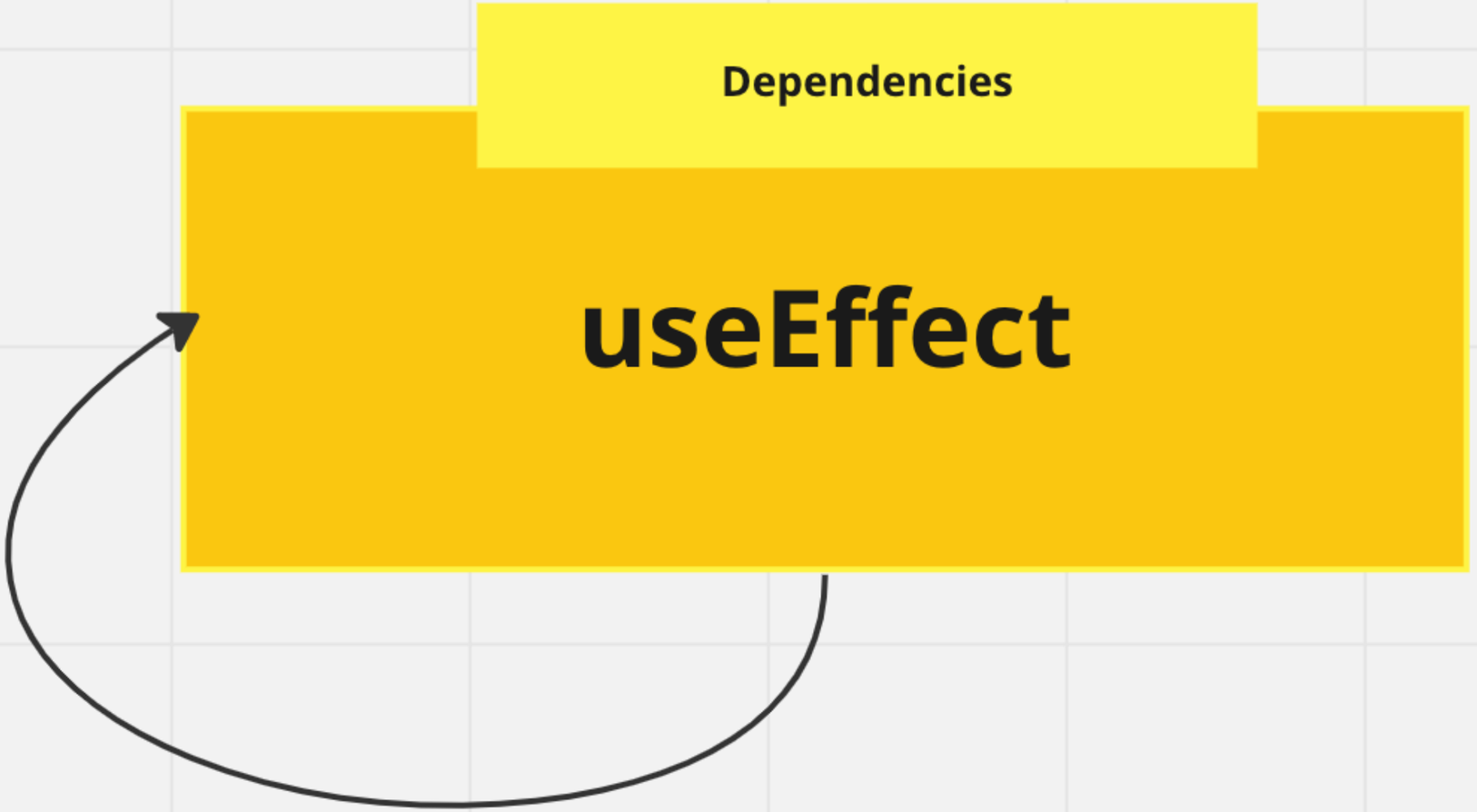
[1] source

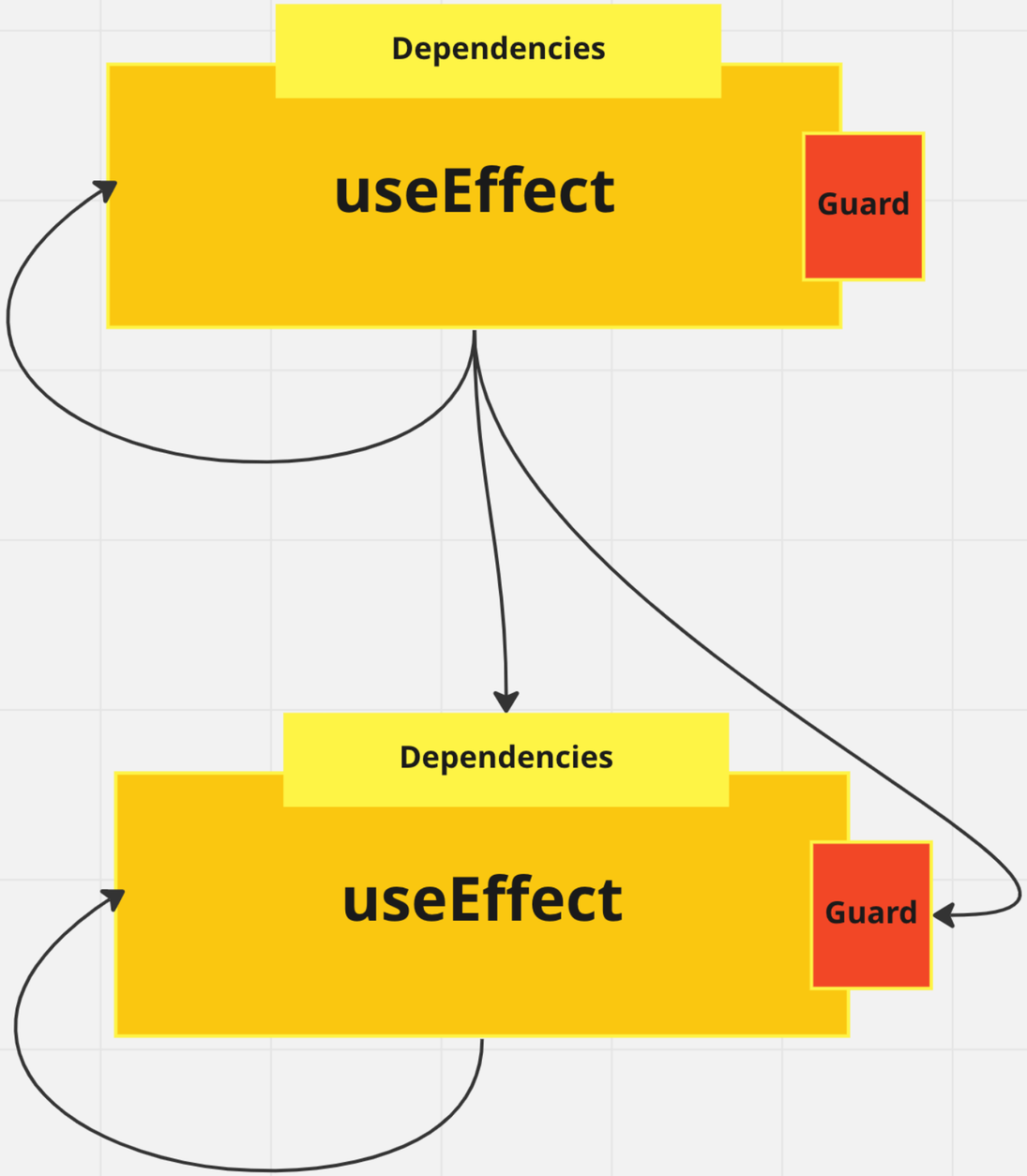
React Query is just useEffect for data fetching on steroids

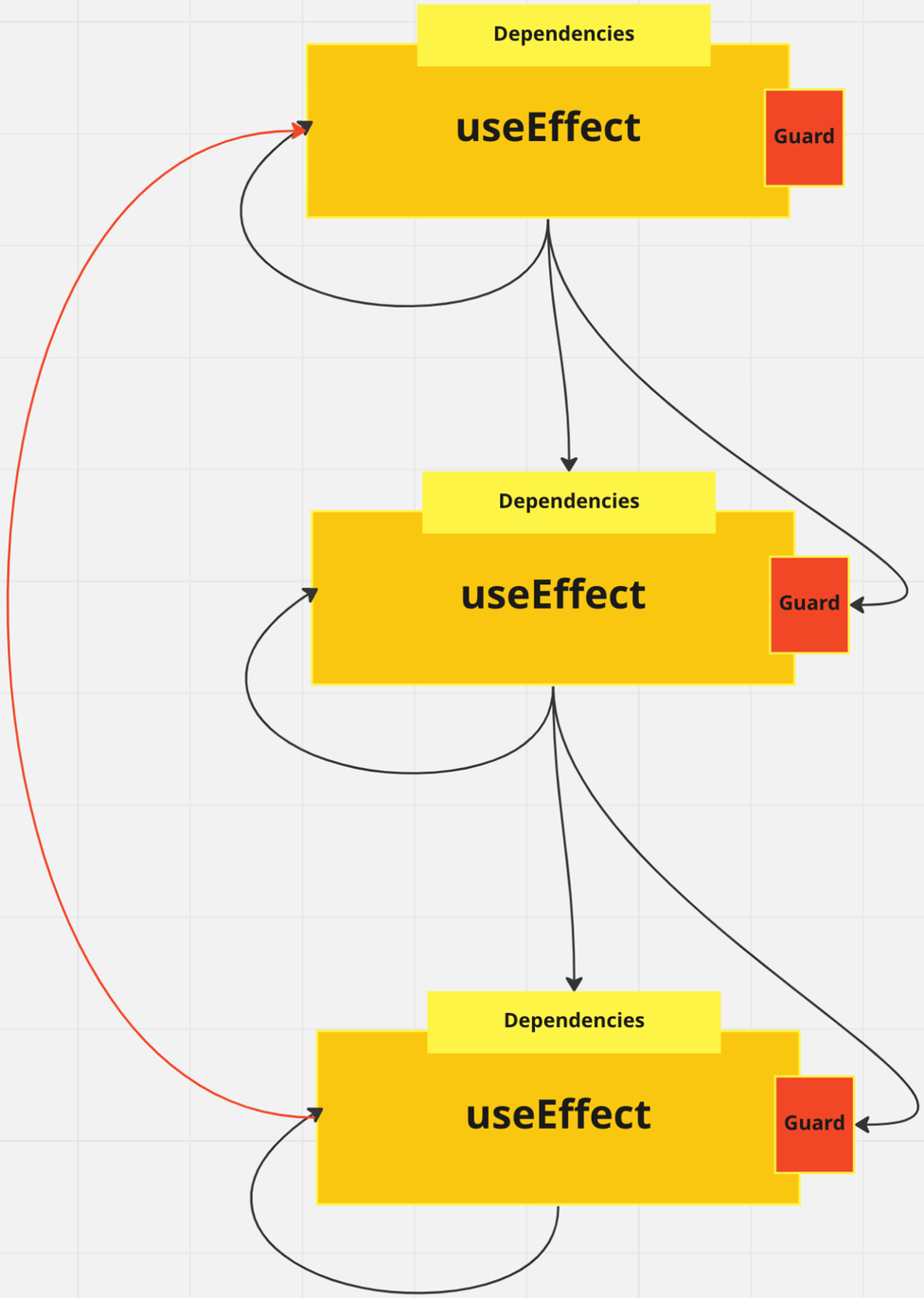
Dependent Effects

```
const [profile, setProfile] = useState<UserProfile | null>(null)
const [userProject, setUserProject] = useState<UserProjects | null>(null)

useEffect(() => {
  getUserProfile(userId).then(profile => setProfile(profile))
}, [userId]);

useEffect(() => {
  if (userProfile) {
    getUserProject(userProfile.projectId).then(project => setUserProject(project))
  }
}, [userProfile]);
```

react how to use use


Dependencies
useEffect


Dependencies
useEffect
Guard
Dependencies
useEffect
Guard


Dependencies
useEffect
Guard
Dependencies
useEffect
Guard
Dependencies
useEffect
Guard

```
const { data: userProfile } = useQuery(['userProfile', userId], () => getUserProfile(userId))

const projectId = user?.projectId

// Then get the user's projects
const { status, fetchStatus, data: projects } = useQuery(
  ['projects', userId],
  () => !!projectId && getUserProject(projectId),
  {
    // The query will not execute until the projectId exists
    enabled: !!projectId,
  }
)
```

Promises compose better than useEffect

```
async function effect() {
  const userProfile = await getUserProfile(userId);
  const userProject = await getUserProject(userProfile.projectId)
}
```

```
const userProfile = await getUserProfile(userId);
const userProjects = await Promise.all(
  userProfile.projectIds.map(projectId => getUserProject(projectId))
)
```

No Promise composition for errors though!

— Effect cancellation

— Effect retries

— Effect race conditions

No loops for hooks

```
userProfile.projectIds.map(projectId => {
  return useEffect(() => {
    getUserProject(userProfile.projectId)
  }, [projectId]);
})
```

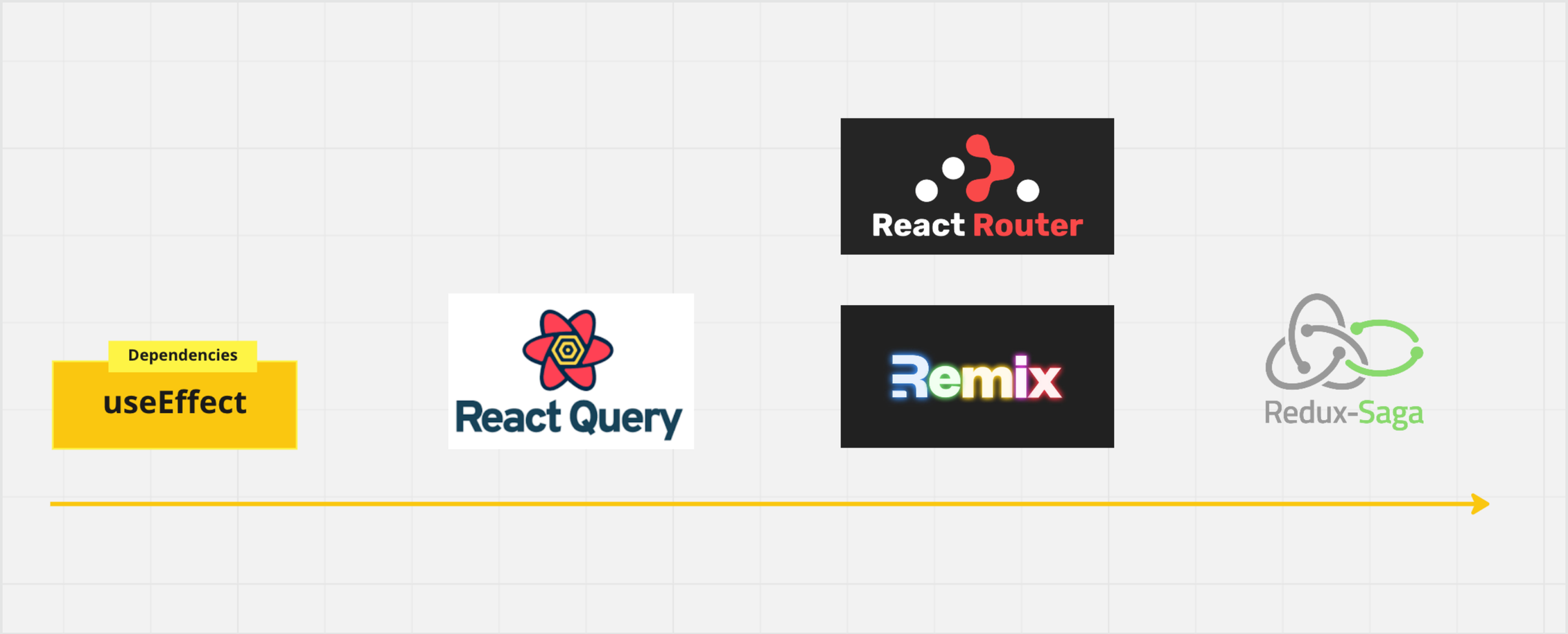
```
const results = useQueries({
  queries: userProfile.projectIds.map(projectId => {
    return {
      queryKey: ['userProject', projectId],
      queryFn: () => getUserProject(projectId),
      staleTime: Infinity
    }
  })
})
```

What else can we use to compose Effects better?

— Observables (rxjs)

— Generators (redux-saga)

Effect management is a sprectrum



useEffect is a **low-level building block**. In practice, it's likely the community will start moving to higher-level Hooks.

— Dan Abramov - A Complete Guide to useEffect (2019)

I personally like using

— useEffect for **synchronizing DOM changes**

— useQuery and useMutation for the **lifecycle of standalone API requests**

— sagas for Effects (tipically network Effects) **orchestration**

Code Issues **Pull requests** 28 Actions Projects Security Insights

RFC: First class support for promises and async/await #229

Open **acdlite** wants to merge 1 commit into `reactjs:main` from `acdlite:first-class-promises`

Conversation 138 Commits 1 Checks 1 Files changed 1

acdlite commented 7 days ago Member

Adds first class support for reading the result of a JavaScript Promise using Suspense:

- Introduces support for async/await in Server Components. Write Server Components using standard JavaScript `await` syntax by defining your component as an async function.
- Introduces the `use` Hook. Like `await`, `use` unwraps the value of a promise, but it can be used inside normal components and Hooks, including on the client.

This enables React developers to access arbitrary asynchronous data sources with Suspense via a stable API.

View the rendered text

```
function Note({id, shouldIncludeAuthor}) {
  const note = use(fetchNote(id));

  let byline = null;
  if (shouldIncludeAuthor) {
    const author = use(fetchNoteAuthor(note.authorId));
    byline = <h2>{author.displayName}</h2>;
  }

  return <div>...</div>
}
```

Dependent effects

```
const userProfile = use(getUserProfile(userId));
const userProject = use(getUserProject(userProfile.projectId))
```

Loop over effects

```
userProfile.projectIds.map(
  projectId => use(getUserProject(projectId))
)
```

— Complexity
— Obscurity and lack of composition
— When writing code: Design for composition
— When picking a tool: How is it gonna help me compose better?

Composing

— Components
— Effects
— Types (?)
— Errors (?)

Jiayi Hu

Front-end developer & consultant

— Twitter: @jiayi_ghu
— Github: @jiayihu
— Email: jiayi.ghu@gmail.com