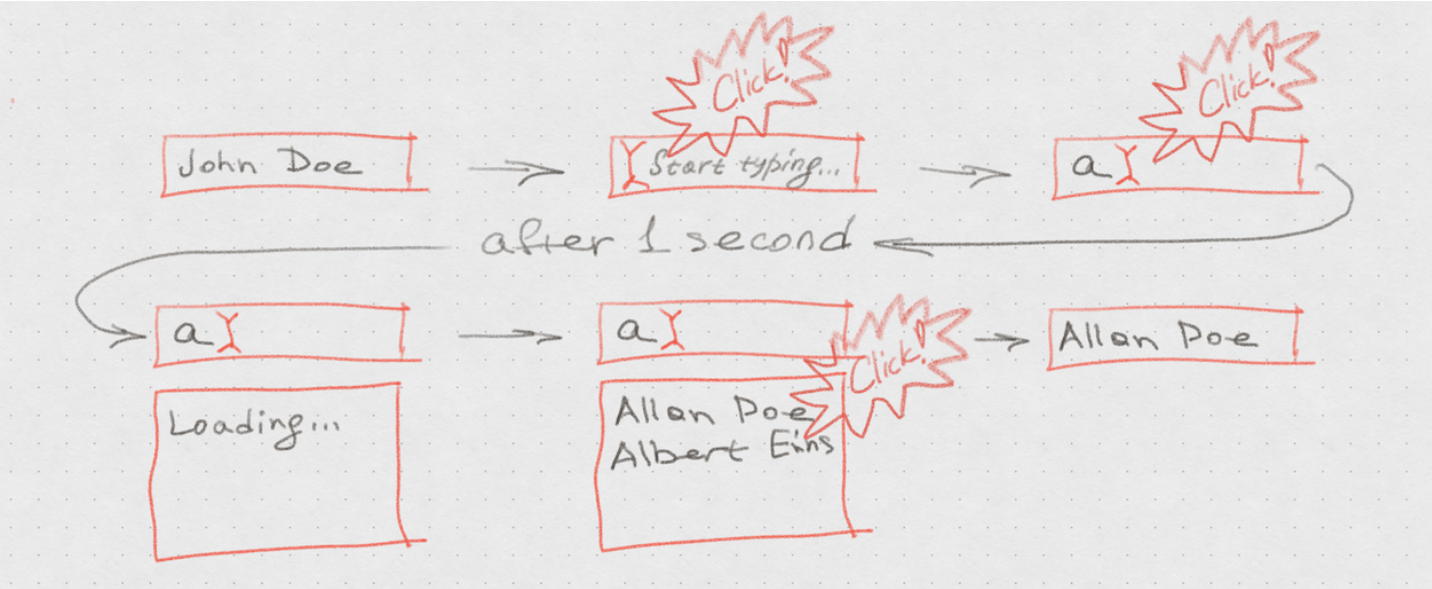


Essential React Hooks Design Patterns

State management and asynchronous I/O

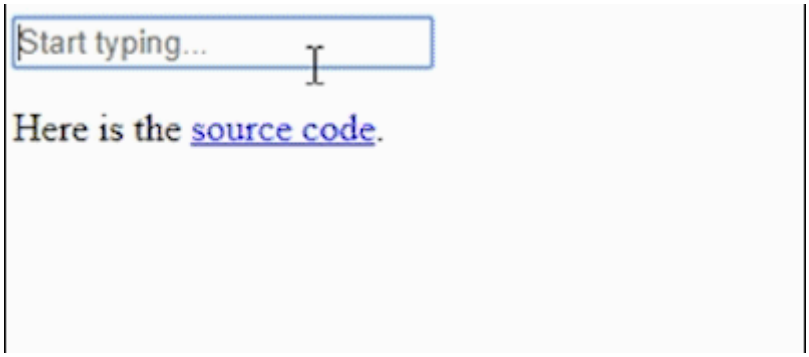
Vlad Balin

May 9 · 11 min read ★

Today we explore the common design patterns of the recently introduced React Hooks API. We are going to cover all the basics you need to create the real application, starting from the state management to the handling of asynchronous I/O. It's best to do in the context of some real-world example, and we will take the "pick user" form control with server-side filtering as such an example. During the exercise, we will identify design problems, explore all the main parts of React Hooks API, and understand how it's going to reduce an amount of code we write daily to pay our bills.

PickUser overview

<PickUser/> form control allows us to pick the user from the database. In its normal state, it just displays the user object. When you click on it and start typing, it will fetch the filtered users from the server and show it as a drop-down list so we can choose the new value. This behavior is quite close to the drop-down list with autocomplete, excepts that the data set can be potentially huge and resides on the server.



<PickUser/> control with server-side filtering

An interesting thing about *<PickUser/>* is that it has a non-trivial state which is not needed outside. Ideally, it should be encapsulated and hidden so we can use this control as easy as the regular *<input/>* inserting it in the same form as many times as we want. Like this:

<PickUser value={user} onChange={setUser} />

It means that we cannot use the global data to store the state and the *<PickUser/>* should rely on its local state to manage all the interactions. This fact alone and the presence of non-trivial side effects and I/O makes *<PickUser/>* quite a challenging task and perfect test-bed for the new React Hooks API.

Stateful components design

Given the fact that we *have to* use local state, we're facing a very practical question. There *must be* some ways to do it both right and wrong. *How to do it right?*

The component state lives as long as the component is mounted. When you decide if some data should or shouldn't be a part of the particular component state, the main question you ask yourself is: "*What is the lifetime of this data?*" The question "*who needs this data*" might seem to be equally important and somewhat easier to answer, but it's really the data lifetime which helps to make an ultimate decision.

So, *what is the lifetime of the data?* If it's the same as a particular component's lifetime, it should be the part of its local state and there are no reasons to expose it or lift it up.

> If a lifetime of the data is the same as the component's lifetime, the data should belong to the component's state. If the data lives longer than the

> component, it should be received as a prop and might be a member of some upper component state. Only the data living longer than any particular component should be stored globally.

Let's take our *<PickUser/>* as an example. The currently selected user can obviously live longer than the *PickUser*, so it's being received as a prop. Does *PickUser* need anything else just to display the user? Nope.

```
const userToString = x => x.name + ' <' + x.email + '>';

const PickUser = ({ user }) => (
  <div>
    <input value={userToString(user)}/>
  </div>
)
```

pickuser.jsx hosted with ♡ by GitHub view raw

Component state and useState() React Hook

When we click on the control, it switches the mode and shows the input to edit the filter. This "mode" should definitely be a part of the state. There is obviously more state than that, but *it's needed during the editing only which is rather rare*, so whatever it is we *just push it down to the <EditUser/> and think about it later.*

> If the state member is not needed all the time, there's an opportunity to push it down to the children components state.

Now, meet the first and most popular React Hook — useState. This hook returns the state value and the function to update it. If you want many state members, just do *useState* multiple times. There's one important thing to remember — **all *useSomething* calls must be done at the top level only. Never do it inside of loops and ifs.**

```
const PickUser = ({ selected, setSelected }) => {
    // Declare the local component's state.
```

```
    const [ editing, setEditing ] = useState( false );

    return (
        <div>
            { editing ?
                <EditUser selected={selected} setSelected={setSelected}
                          close={() => setEditing( false )}/>
            :
                <input value={ userToString( selected ) }
                       onClick={ () => setEditing( true ) }/>
            }
        </div>
    )
}
```

pick-user-state.js hosted with ♡ by GitHub view raw

Writing Custom React Hooks

<EditUser/> will modify the user, so we need to pass both the *selected* and the *setSelected()* through. Passing a lot of state elements around might quickly create a mess, so let's repack *value* and *setValue()* to an object which can be passed as a single value. *Such an object is called the Link [to the state] and represents the mutable reference to the state element.*

But how can we do that in a way that it would actually be suitable to use?

```
function useLink( init ){
   const [ value, set ] = useState( init );
   // It can be a class with useful methods, like this one:
   // https://github.com/VoliJS/NestedLink/blob/master/valuelink/src/link.ts
   // But we just use the plain object here to illustrate an idea.
   return { value, set };
}

const PickUser = ({ $selected /* link to some upper component state */ }) => {
    // Declare the local component's state as a link.
    // Now, state elements are easy to distinguish and pass around.
    const $editing = useLink( false );

    return (
      <div>
         { $editing.value ?
             <EditUser $selected={$selected /* just pass it through */}
                       close={() => $editing.set( false )}/>
          :
```

```
            <input value={ userToString( $selected.value ) }
                   onClick={ () => $editing.set( true ) }/>
          }
        </div>
      )
  }
```

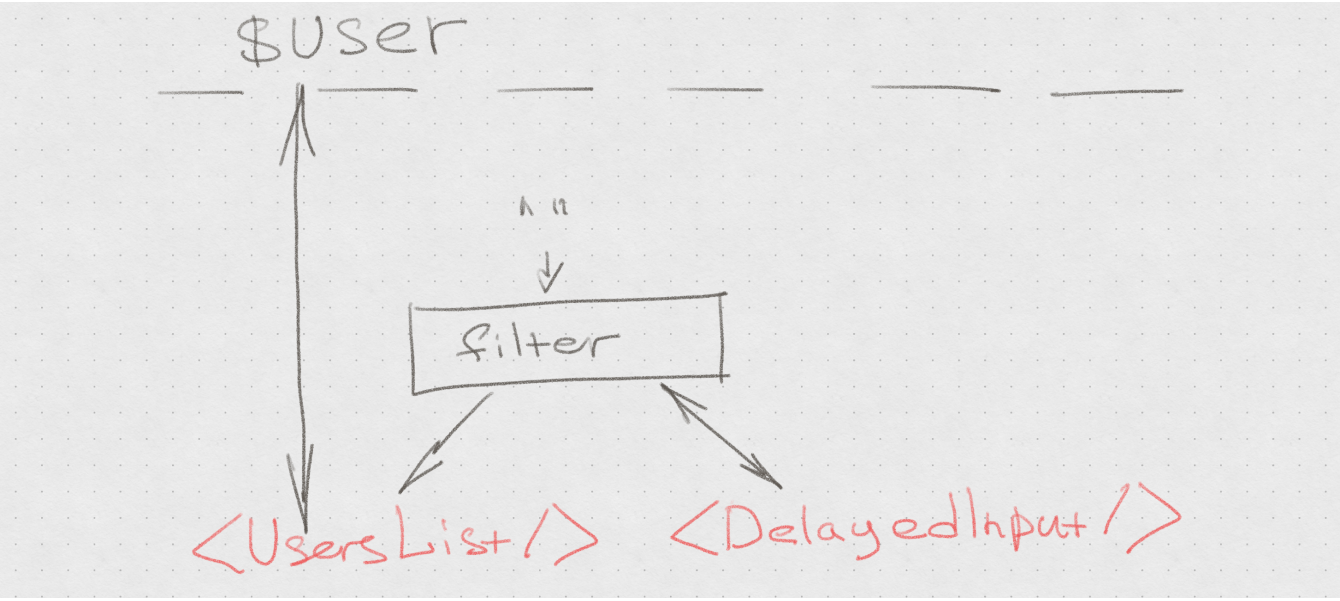
pickuser2.jsx hosted with ♡ by GitHub view raw

Here is our first custom hook — *useLink*! A "custom hook" is just a function with other React hook calls inside. **It's okay to use hooks in your functions, just start their names with "use" so it will be clear that they are also "hooks".**

> Links are "writable props" representing the writable references to some component's state. Links simplify passing the upper state down to children.

Shared component's state

<EditUser/> will handle the editing. The only part of the state with a lifetime equal to the one of the *EditUser* is the *filter*. We will need the *<input/>* to change the *filter*, and another component *<UsersList/>* which will fetch and show the filtered list of users from the server whenever its *filter* prop changes.



We should, however, prevent the fetch from happening too frequently while our user types the filter. Therefore, we can't use plain *<input/>* and need a similar component which will delay changes up to the moment the user stopped typing. Let's call it *<DelayedInput/>*.

```jsx
export const EditUser = ({ $selected, close }) => {
    const $filter = useLink('');

    return (
        <>
            <DelayedInput autoFocus
                $value={ $filter }
                placeholder="Start typing..."
                onBlur={ close } />

            { $filter.value ?
                <UsersList filter={$filter.value} $selected={$selected} />
            : void 0 }
        </>
    );
}
```

ac-edit-user.jsx hosted with ♡ by GitHub view raw

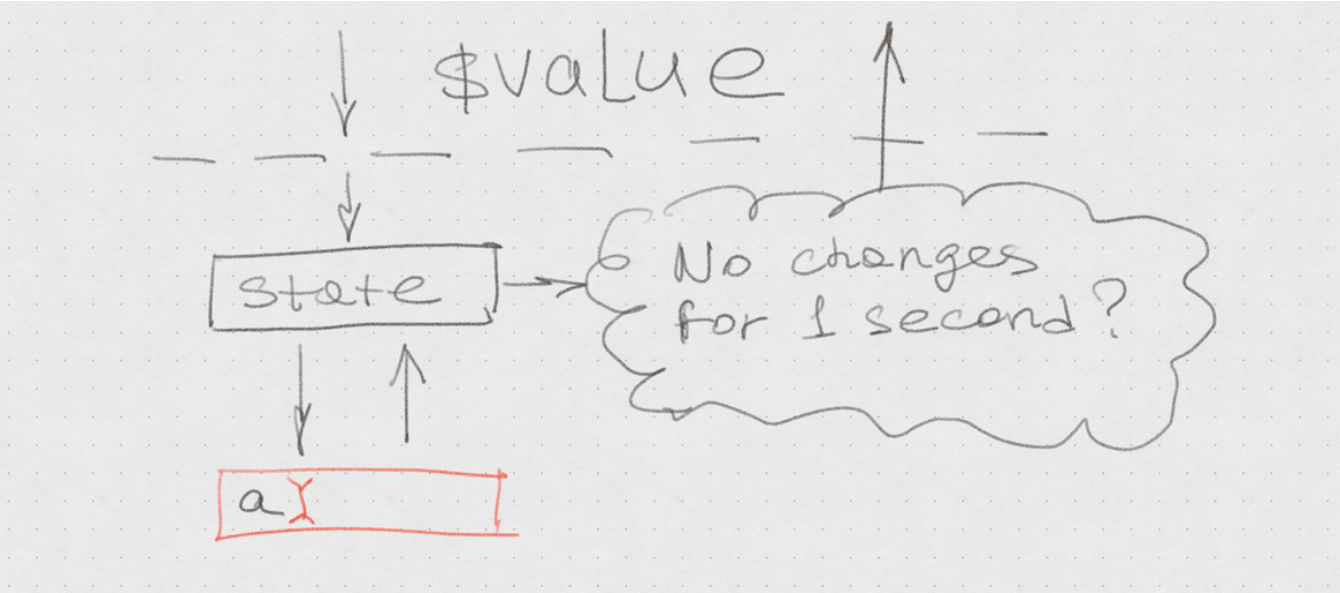
Here we have an example of the situation when two components operate with a *shared state*. Such a state, indeed, has to be lifted up to the closest common root of the interacting components. *If this state is not used all the time*, there's an opportunity to rethink the component's structure and create the new child component which would encapsulate such a state without the "data lifetime rule" violation. Opportunity is not an obligation, but if you will use it might make the system simpler.

> The state which is shared between several components should be lifted up to the closest common root component. Watch for the "data lifetime rule" when you're doing that, consider refactoring when necessary.

Now when we narrowed the problem down to two different components, we will deal with an I/O problem and the problem of the delayed input separately. And that's where we will need the full power of React Hooks.

<DelayedInput />

This control should behave similar to the regular *input*, but delay the value change for a given amount of milliseconds. If the user types again during the delay it must be extended. So, *DelayedInput* receives its value from the parent, but it must keep the changes private until the delay will pass. If the parent will change the value for some other reason, though, the local value must be changed.



It means, we need a local state which is one-way synchronized with the *$value* prop. Let's do this synchronization, wrapping it in a custom hook.

State synchronization and useEffect() React Hook

It's time to meet React's *useEffect()* hook. *useEffects*() executes the given function after the render, and it will only do it when the elements of the array in its second argument change. That's exactly what we need. We will use to track the *$value* changes and update the state, and we will wrap this logic into the custom hook.

```
function useBoundLink( source ){
    // If the source is another Link, extract the value out of it.
    const value = source instanceof Link ? source.value : source,
          link = useLink( value );

    // If the value changes, execute link.set( value ) after the render.
    useEffect(() => link.set( value ), [ value ]);
```

```
8
9       return link;
10  }
```

use-bound-link.js hosted with ♡ by GitHub view raw

Bound state link differs from a plain link in a single aspect: whenever its default value changes, it changes the underlying state, so we can just pass the prop value as default and it will do the job. That's it, state synchronization problem solved.

> useEffect() React hook can track changes so you can easily attach reactions to props changes after render. That's what you do in place of old componentWillReceiveProps.

Now, let's touch the power of *Link* pattern a bit. Manual *onChange* event handlers never were the most exciting part of React. What if the *Link* wouldn't be a plain object, but a class with some useful methods? Could it save us some typing *generating the onChange event handler* for us? The answer is "yes" if you're using *useLink hooks* from the *NestedLink library*. Let's do that.

```
1  export const DelayedInput = ({ $value, timeout = 1000, …props }) => {
2   const $inputValue = useBoundLink( $value );
3   // TODO: How to sync the state back?
4   return <input {…$inputValue.props} {…props}/>;
5  }
```

delayed-input.jsx hosted with ♡ by GitHub view raw

Synchronizing state back to props

$inputValue.props.onChange handler updates the *$inputValue* state while the user is typing. We want it to do so, but we also want to set up the timer on each change, and when it will fire we will update the parent state. If another state update will happen before that, we will cancel the timer and create the new one.

Without links, we would just put this logic to the input's *onChange*. With NestedLink, we have an additional option to attach the *onChange* listener directly to the link. Let's do that, assuming that we have something similar to a Lodash _.throttle function.

```jsx
export const DelayedInput = ({ $value, timeout = 1000, ...props }) => {
    const $inputValue = useBoundLink( $value )
        .onChange(
            useThrottle(
                x => $value.set( x ),
                timeout,
                [ $value.value ]
            )
        );

    return <input {...$inputValue.props} {...props}/>;
}
```

delayed-input-3.jsx hosted with ♡ by GitHub view raw

Unfortunately, we can't just use *_.throttle, it won't work for a variety of reasons*. In order to cancel the previous timer we have to know its *id*, and it sounds like another component state which lodash is totally unaware of.

> *Someone might argue that we may rely on* ***useCallback hook*** *to make underscore/lodash _.throttle function work.* ***No, we can't.*** *We need to cancel the timer when component will unmount to prevent an exception, and lodash is unaware of mounts and unmounts.*

So, we will create the custom *useThrottle hook,* and it's time to meet the useRef React Hook.

useRef and cleanup effects

useThrottle take the function and produce another function, which will delay the call for a given amount of milliseconds. We want to store timer *id* in the component's state in order to cancel it when necessary, but *we don't want the component to render when we change it*. We want it to behave like a raw class member in the old class *Component*. That's the situation the useRef React Hook is designed for.

> useRef React hook has the same meaning as a regular class member of the React class component

Also, *useThrottle* takes a strange third argument [$value.value]. That thing is needed to handle the race condition: what if the parent's value will change while we're typing? We will assign it back with an old value because we still have a timer scheduled. Instead, we need to cancel it, and we are using *useEffect* hook *cleanup* for that.

> Clean-up is the function returned from the useEffect body, which is called right before the next useEffect body call and on unmount.

```
function useThrottle( fun, timeout, changes = [] ){
    // Create the mutable local ref to store timer.
    const timer = useRef( null );

    function cancel(){
        if( timer.current ){
            clearTimeout( timer.current );
            timer.current = null;
        }
    }

    // Cancel the timer when the given values change or the component will unmount.
    useEffect( () => cancel, changes );

    // Return the throttled version of the function.
    return function( ...args ){
        cancel();

        // Save the timer to the ref, so it can be cancelled.
        timer.current = setTimeout(()=>{
            timer.current = null;
            fun.apply( this, args );
        }, timeout );
    }
}
```

use-throttle.js hosted with ♡ by GitHub view raw

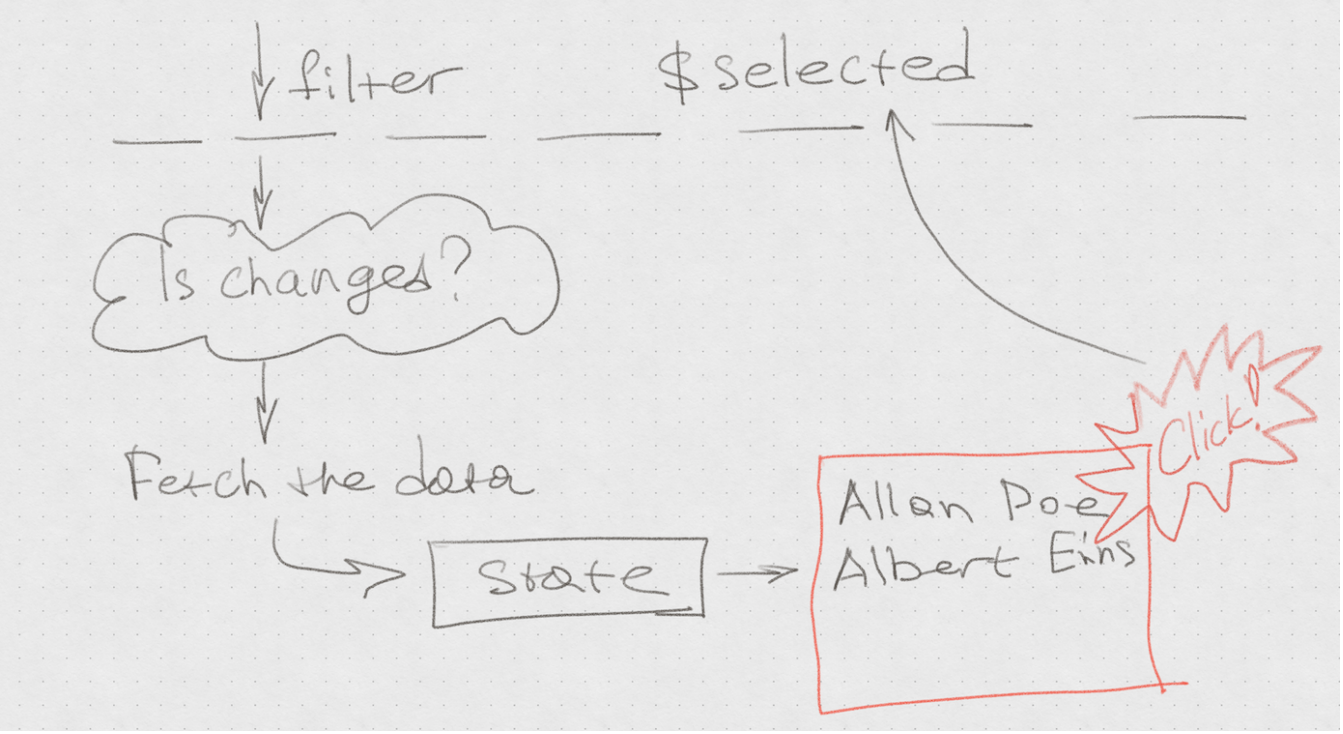
Whatever you return from the function within the *useEffect* is treated as a *“cleanup”* function. It will be called when the *useEffect* react on change and on unmount, right before the useEffect body will be called. *The difference between cleanup and body is that cleanup is bound to values from the previous useEffect body call.* So, we don’t have to worry about another race condition when the user will close our control but there was the delayed update to the dead state.

DelayedInput in done. There are two news — good, and also good. First, now we know everything we need about hooks to deal with asynchronous I/O. Second, you don’t

need to repeat this exercise ever again. I added *useThrottle* and *DelayedInput* to the linked-controls npm package, so if you will ever need them just grab them from there.

<UsersList/>

<UsersList/> accepts the *filter* as a prop and has to react on its change fetching the filtered data from the server and displaying it. Thanks to the *<DelayedInput/>* it won't happen too frequently.



We already know that the *useEffect* can be used to track the props changes. But we would like to use an *async* and *await* to fetch the data, and here's the catch — *useEffect* assumes that *whatever you return from within is a cleanup function.* An async function always returns a promise. Also, it would be useful to know whenever the I/O operation is pending to indicate that something is "loading…". Let's wrap this logic in a custom *useIO* hook.

> You can't use async and await inside of useEffect. But you can create a custom hook to work it around.

```
const UsersList = ({ filter, $selected }) => {
    // $users can be modified by async function after the component is unmounted.
```

```
    // We have to do something to prevent an exception. Let's do it in this custom hook.
    const $users = useSafeLink([]);

    // It's useful to know if there's an I/O peding. Another custom hook.
    const ioComplete = useIO( async () => {
        // This thing can happen after unmount.
        $users.set( await fetchUsers( filter ) );
    }, [ filter ]);

    return (
        <ul className="users-suggestions">
            { ioComplete ? $users.value.map( user => (
                <li key={user.id}
                    className={ $selected.value && $selected.value.id === user.id ? 'selected'
                    onMouseDown={ () => $selected.set( user ) }
                >
                    { userToString( user ) }
                </li>
            )) : 'Loading...' }
        </ul>
    )
}
```

ac-users-list.jsx hosted with ♡ by GitHub view raw

As we wrote the code, we noticed another small problem — there's an exception possible when there's a pending I/O while the component will unmount. Let's repeat it with big letters:

> You can't just update your state when I/O is completed. The component might be unmounted and you get an exception.

That sounds rather disturbing, but I can assure you — it's not a problem for us with our just obtained magnificent custom hook programming skills. Let's not panic, and see what we can do.

Checking if the component is mounted

To prevent an exception, we need to know if the component is mounted when executing the link.set. How can we possibly know that within the functional

component? If only we would have something similar to *componentWillUnmount*… And here it is! *useEffect with the cleanup* is to the rescue.

```
export function useIsMountedRef(){
    // We need something similar to the plain mutable class member.
    const isMounted = useRef( true );

    // And, we need something similar to componentWillUnmount.
    useEffect( () => {
        // Whatever we return is a cleanup effect.
        return () => { // <- componentWillUnmount
            isMounted.current = false
        }
    }, []); // [] never changes, so the "cleanup" function will be fired on unmount only.

    return isMounted;
}
```

use-is-mounted-ref.js hosted with ♡ by GitHub view raw

Now we touched, probably, one of the most important *useEffect* use cases:

> useEffect(whenDidMount, []) behaves as componentDidMount, and it's cleanup effect as a componentWillUnmount.

Prevent state updates when the component is unmounted

Okay. Now this task seems to be trivial. Let's do it once in another custom hook and forget about it.

```
export function useSafeLink( initialState ){
    const $value = useLink( initialState ),
            isMounted = useIsMountedRef();

    const { set } = $value;
    $value.set = x => isMounted.current && set( x );
    return $value;
}
```

use-safe-link.js hosted with ♡ by GitHub view raw

> All asynchronous state updates from I/O functions must be guarded agains the possible component unmount.

I/O with async-await and loading indicator

useIO will return true when the promise is resolved, and false otherwise. This small requirement becomes particularly tricky to implement if we take into account that there might be *filter* change before the previous I/O operation is finished. It means that there might be more than one unresolved I/O promise at a time. To address that, we will maintain the counter of unresolved I/O promises, and if it's zero we decide that I/O is finished.

```
function useIO( fun, condition = [] ) {
    // Counter of open I/O requests. If it's 0, I/O is completed.
    // Counter is needed to handle the situation when the next request
    // is issued before the previous one was completed.
    const $isReady = useSafeLink( null );

    useEffect(()=>{
        // function in set instead of value to avoid race conditions with counter increment.
        $isReady.set( x => ( x || 0 ) + 1 );

        fun().finally(() => $isReady.set( x => x - 1 ));
    }, condition);

    // null is used to detect the first render when no requests issued yet
    // but the I/O is not completed.
    return $isReady.value === null ? false : !$isReady.value;
}
```

use-io.js hosted with ♡ by GitHub view raw

Whoops, we're using the component state here which is modified in *promise.finally()!* Okay, so change *useLink* to *useSafeLink*. Not a big deal. Not anymore.

. . .

It's hard to believe, but we finished. Here it is, working, and here is the source code. During this exercise, we used all of the important React hooks and now we have a

general understanding of what new Hooks API is capable of and know its basic application patterns. Also, we learned the basics of the design and decomposition of the stateful components, and (I hope) understood what these new hooks really are.

> React Hooks are extremily powerful stateful mixins for React functional components which can encapsulate complex design patterns in a quite concise and elegant way.

Also, we created a bunch of quite useful custom hooks during this exercise. It would be a shame to write them again and again, so I added them to the NestedLink library. Happy coding.

```
// npm install valuelink linked-controls --save-dev

// That's what you need to start.
import { useLink } from 'valuelink'

// Hooks used in DelayedInput, and the DelayedInput himself.
import { useBoundLink } from 'valuelink'
import { useThrottle, DelayedInput } from 'linked-controls'

// Hooks used in UsersList
import { useIsMountedRef, useSafeLink, useIO } from 'valuelink'

// And, just in case you'll need it...
import { useSafeBoundLink } from 'valuelink'

```

ac-imports.jsx hosted with ♡ by GitHub view raw

JavaScript React React Hook Tutorial

About Help Legal