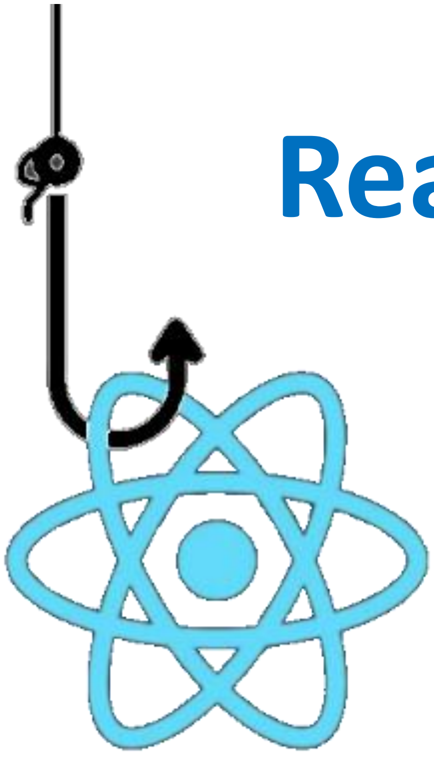
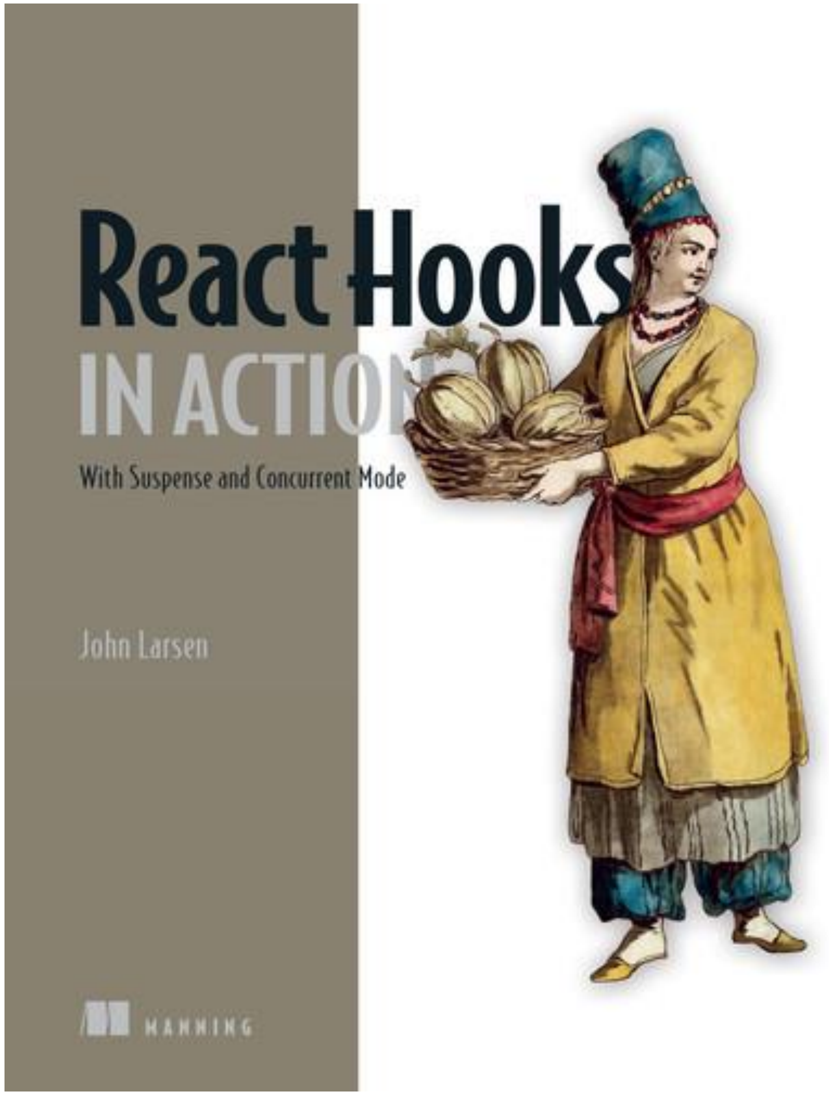


React Hooks

Outline

Slides are based on



https://learning.oreilly.com/library/view/react-hooks-in/9781617297632/

What is Hook?

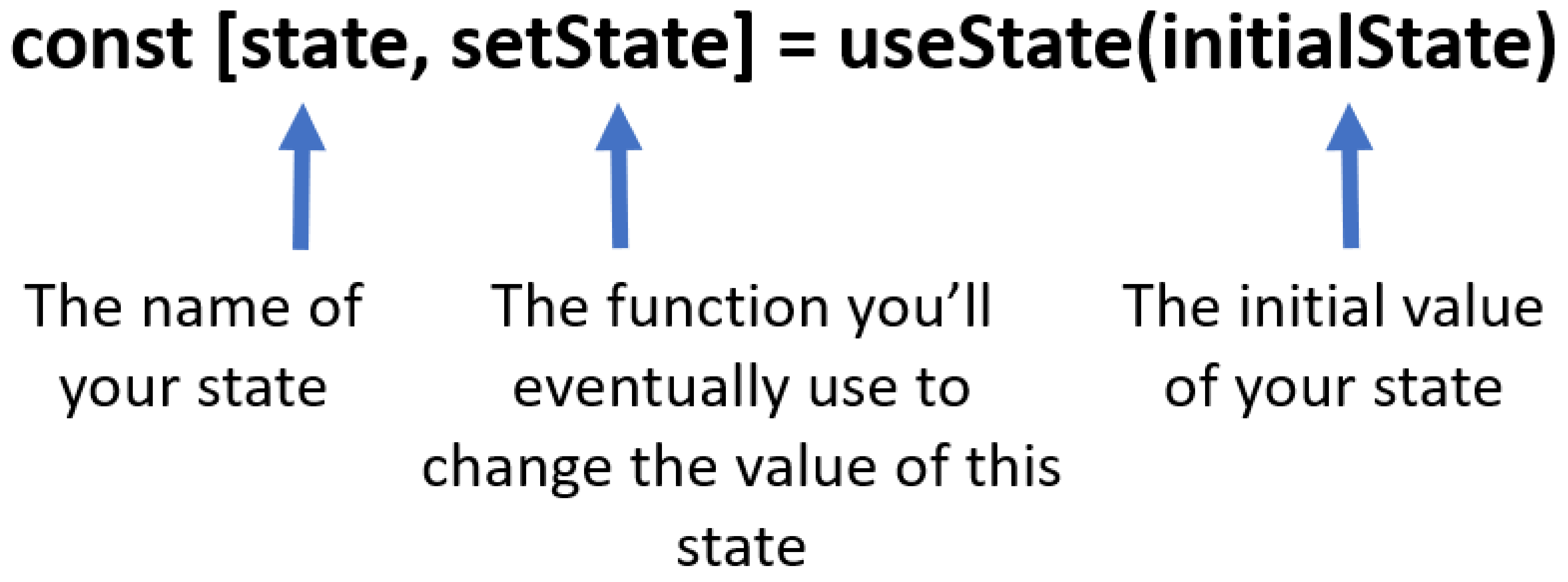
- A Hook is a special function that lets you **hook** into React features such as state and lifecycle methods
- There are 3 rules for hooks:
  - Hooks can only be called inside React function components.
  - Hooks can only be called at the top level of a component.
  - Hooks cannot be conditional

Common Hooks

Back

useState: creates a state variable

- Used for basic state management inside a component



```
const [state, setState] = useState(initialState)
```

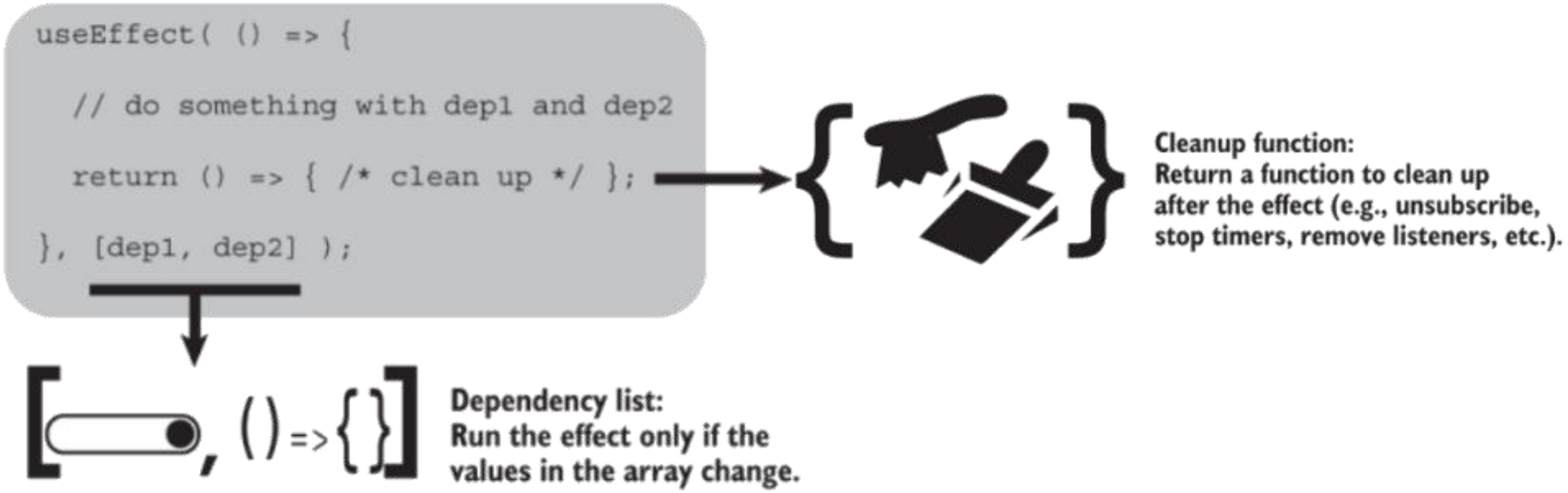
The name of your state

The function you'll eventually use to change the value of this state

The initial value of your state

useEffect

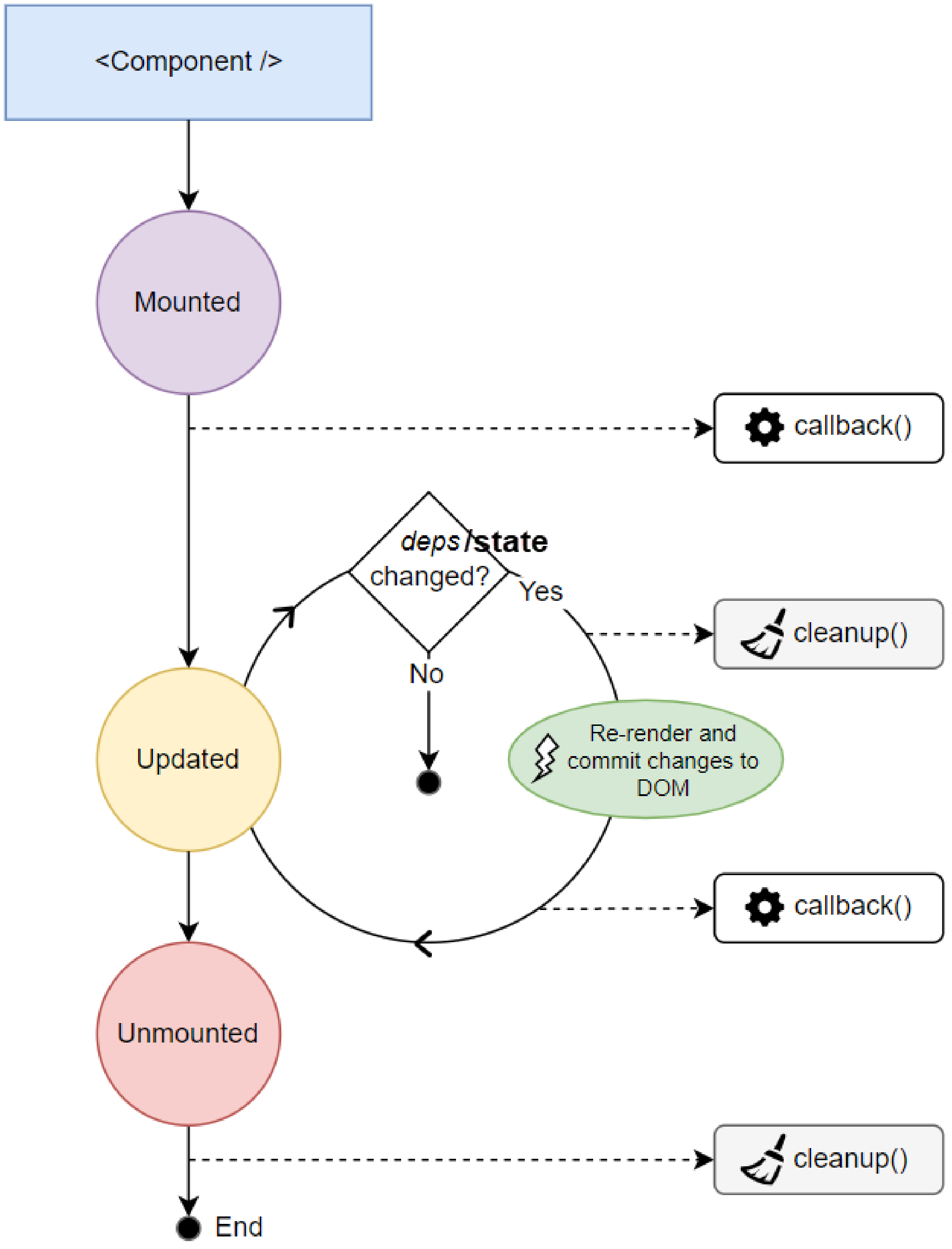
- For doing stuff when a component is mounts/unmounts/updates
- Ideal for fetching data when the component is mounted



```
useEffect( () => {
  // do something with dep1 and dep2
  return () => { /* clean up */ };
}, [dep1, dep2] );
```

Cleanup function:
Return a function to clean up after the effect (e.g., unsubscribe, stop timers, remove listeners, etc.).

Dependency list:
Run the effect only if the values in the array change.



A) After initial rendering, `useEffect()` invokes the callback having the side-effect. cleanup function is not invoked

B) On later renderings, before invoking the next side-effect callback, `useEffect()` invokes the cleanup function from the previous side-effect execution (to clean up everything after the previous side-effect), then runs the current side-effect

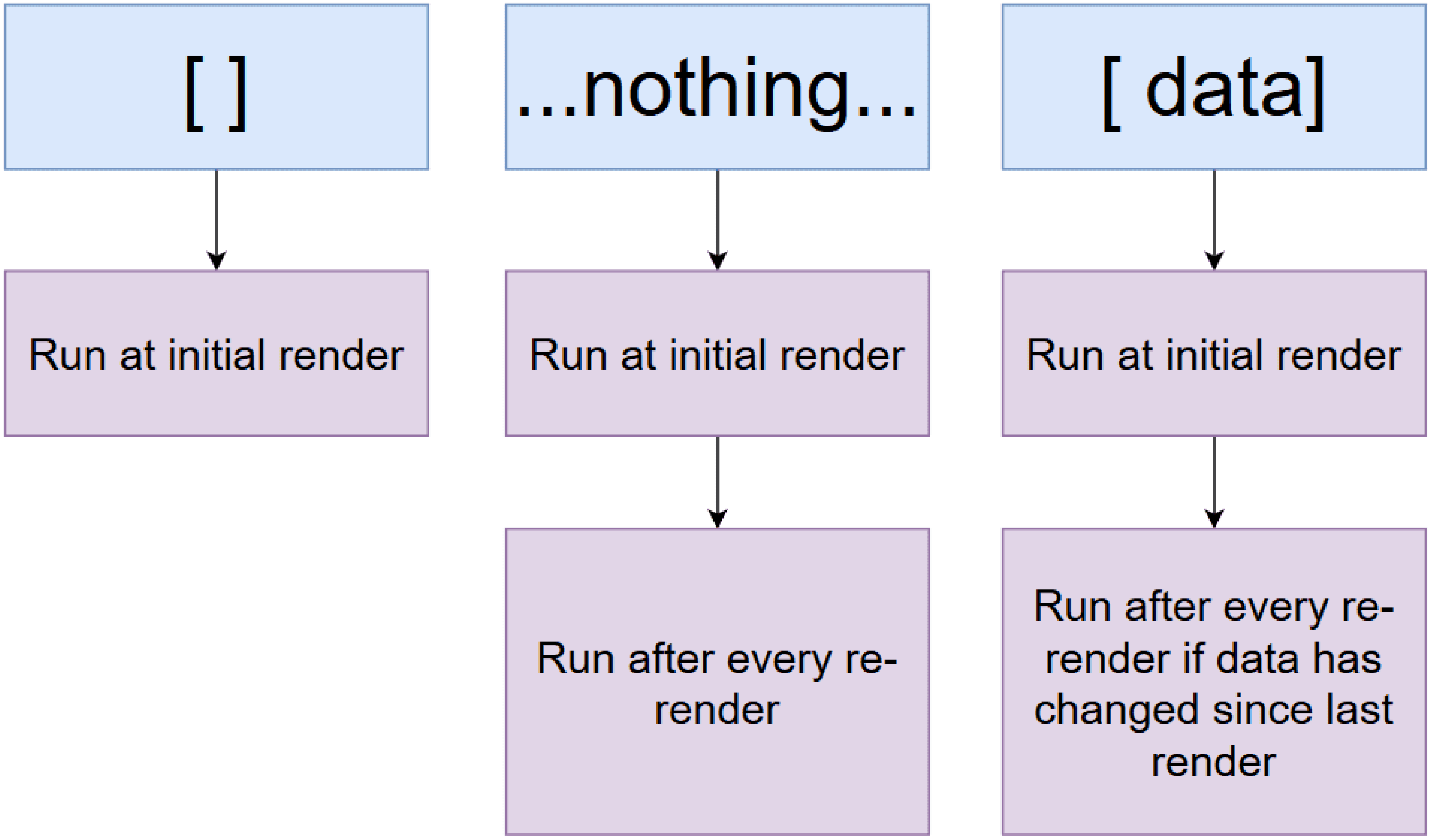
C) Finally, after unmounting the component, `useEffect()` invokes the cleanup function from the latest side-effect

Common side effects

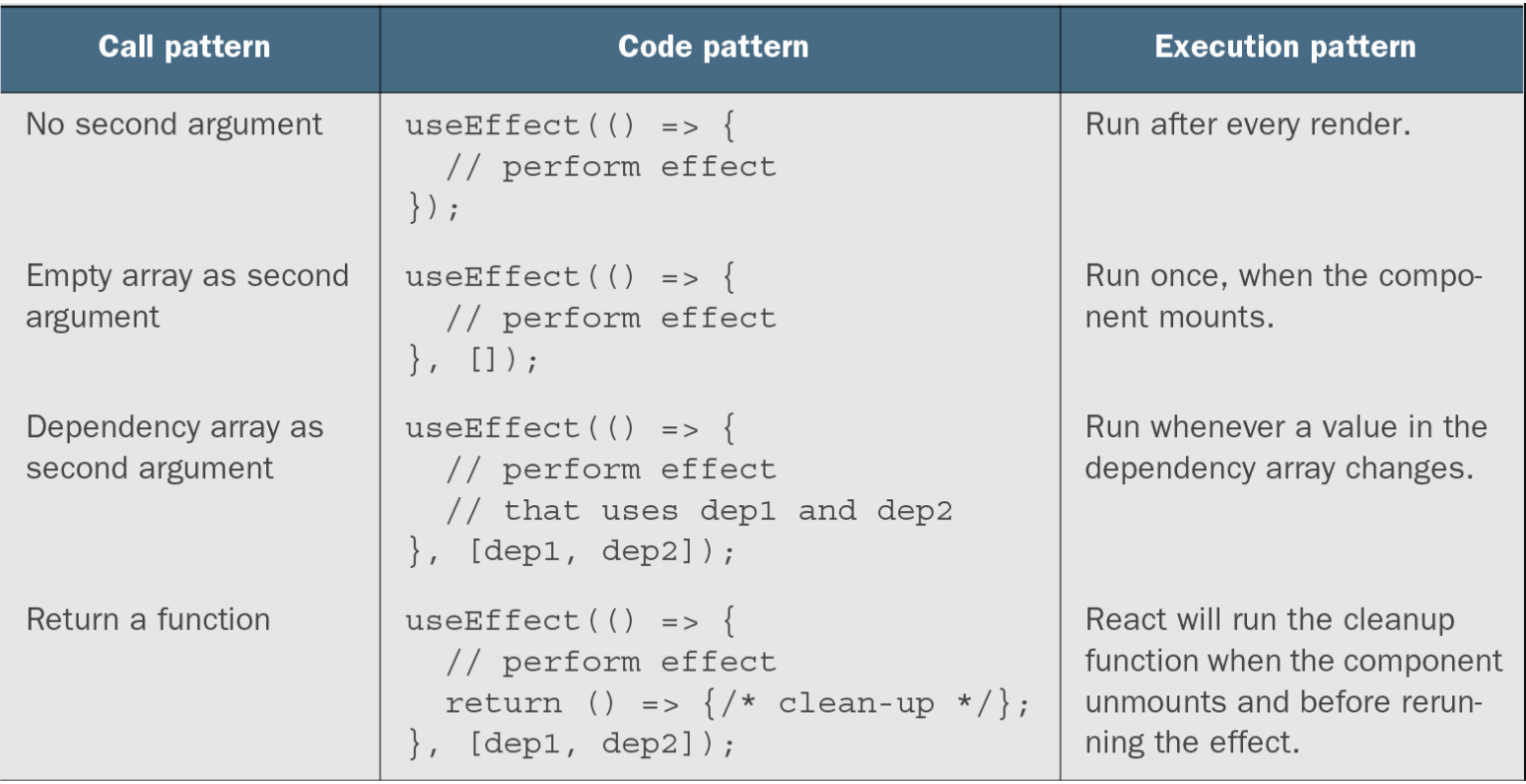
Common side effects include:

- Setting the page title imperatively
- Working with timers like `setInterval` or `setTimeout`
- Logging messages to the console or other service
- Fetching data or subscribing and unsubscribing to services
- Setting or getting values in local storage

useEffect - 2nd argument


[ ]
...nothing...
[data]
Run at initial render
Run at initial render
Run at initial render
Run after every re-render
Run after every re-render if data has changed since last render

Use cases for the useEffect hook



Call pattern	Code pattern	Execution pattern

No second argument	`useEffect(() => {`
	`  // perform effect`
	`});`
		Run after every render.

Empty array as second argument	`useEffect(() => {`
	`  // perform effect`
	`}, []);`
		Run once, when the compo-nent mounts.

Dependency array as second argument	`useEffect(() => {`
	`  // perform effect`
	`  // that uses dep1 and dep2`
	`}, [dep1, dep2]);`
		Run whenever a value in the dependency array changes.

Return a function	`useEffect(() => {`
	`  // perform effect`
	`  return () => {/* clean-up */};`
	`}, [dep1, dep2]);`
		React will run the cleanup function when the component unmounts and before rerun-ning the effect.

useEffect – Executes code during Component Life Cycle

- **Initialize state data** when the component loads

```
useEffect(() => {
    async function fetchData() {
        const url = "https://api.github.com/users";
        const response = await fetch(url);
        setUsers( await response.json() ); } // set users in state
         fetchData();
}, [ ]); // pass empty array to run this effect once when the component is first mounted to the DOM.
```

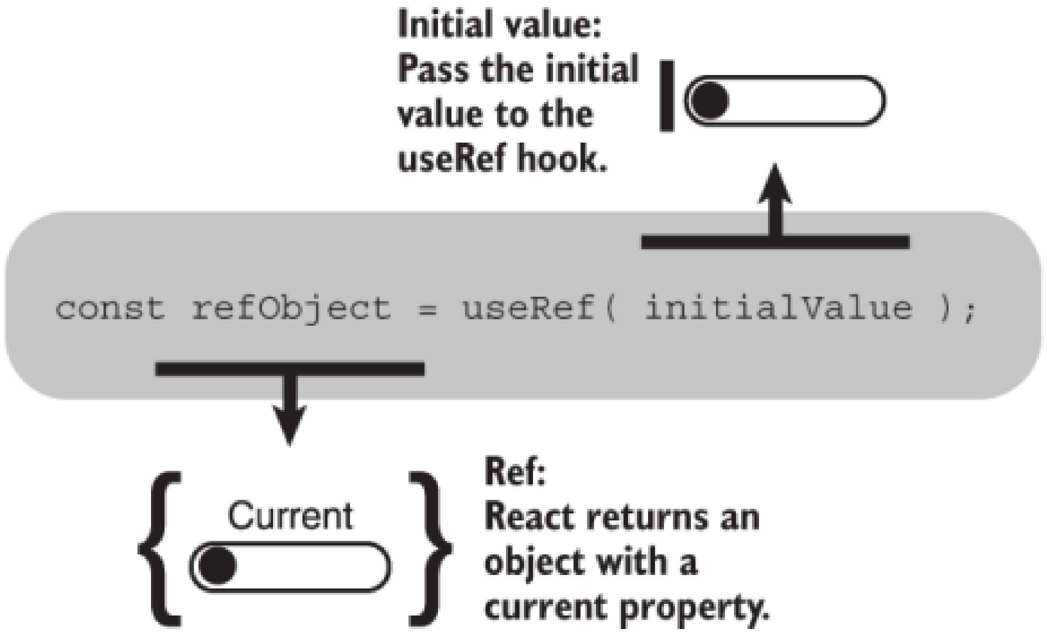
- **Executing a function every time a state variable changes**

```
useEffect(() => {
    async function fetchData() {
        const url = `https://hn.algolia.com/api/v1/search?query=${query}`;
        const response = await fetch(url);
        const data = await response.json();
        setNews(data.hits);
    }
    fetchData();
}, [query]);
```

If 2nd parameter is not set, then the useEffect function will run on every re-render

useRef

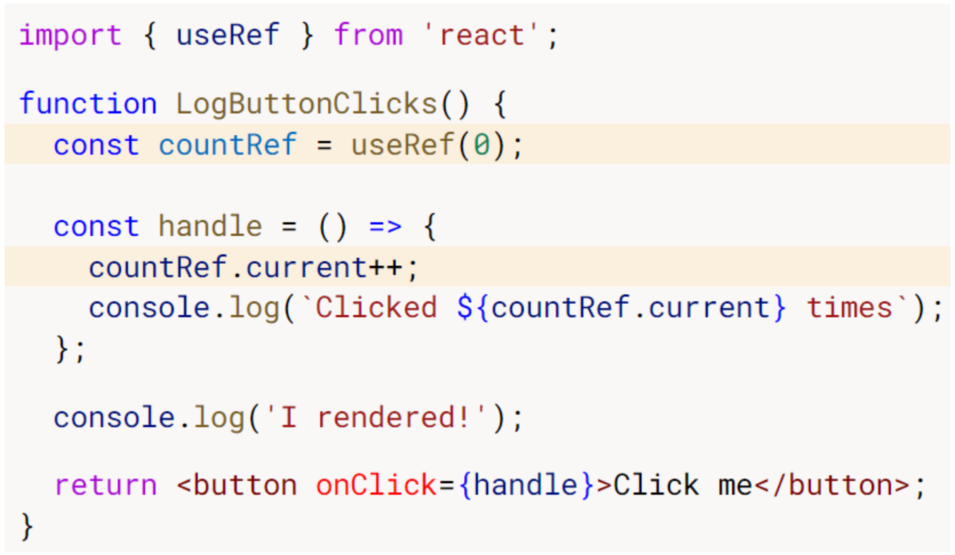
- useRef() hook to create **persisted mutable values** as well as directly **access DOM elements** (e.g., focusing an input)
  - The value of the reference is persisted (stays the same) between component re-renderings;
  - Updating a reference doesn't trigger a component re-rendering.



Initial value:
Pass the initial value to the useRef hook.
const refObject = useRef(initialValue);
Current
Ref:
React returns an object with a current property.

useRef for Mutable values

- useRef(initialValue) accepts one argument as the initial value and returns a reference. A reference is an object having a special property current



```
import { useRef } from 'react';

function LogButtonClicks() {
  const countRef = useRef(0);

  const handle = () => {
    countRef.current++;
    console.log(`Clicked ${countRef.current} times`);
  };

  console.log('I rendered!');

  return <button onClick={handle}>Click me</button>;
}
```

- `reference.current` accesses the reference value, and `reference.current = newValue` updates the reference value
- The value of the reference is persisted (stays the same) between component re-renderings
- Updating a reference doesn't trigger a component re-rendering

useRef for accessing DOM elements

- useRef() hook can be used to access DOM elements

```
import { useRef, useEffect } from 'react';

function InputFocus() {
  const inputRef = useRef();

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return (
    <input
      ref={inputRef}
      type="text"
    />
  );
}
```

- Define the reference to access the element

`const inputRef = useRef();`

- Assign the reference to **ref** attribute of the element:

` <input  ref={inputRef} />`

- After mounting, `inputRef.current` points to the DOM element

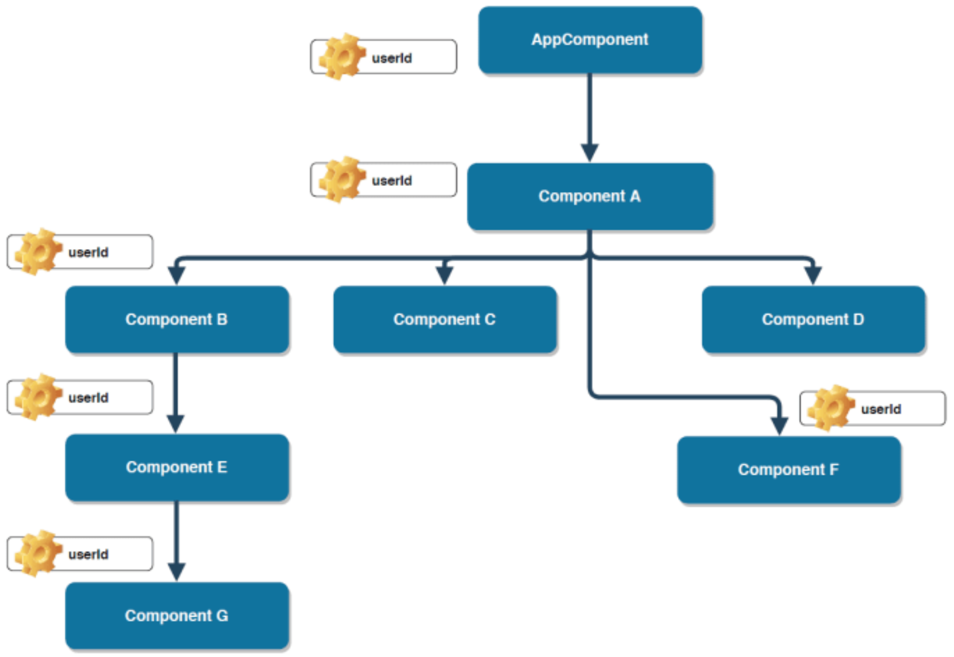
=> In this example, we access the input to focus on it when the component mounts. After mounting we call `inputRef.current.focus()`

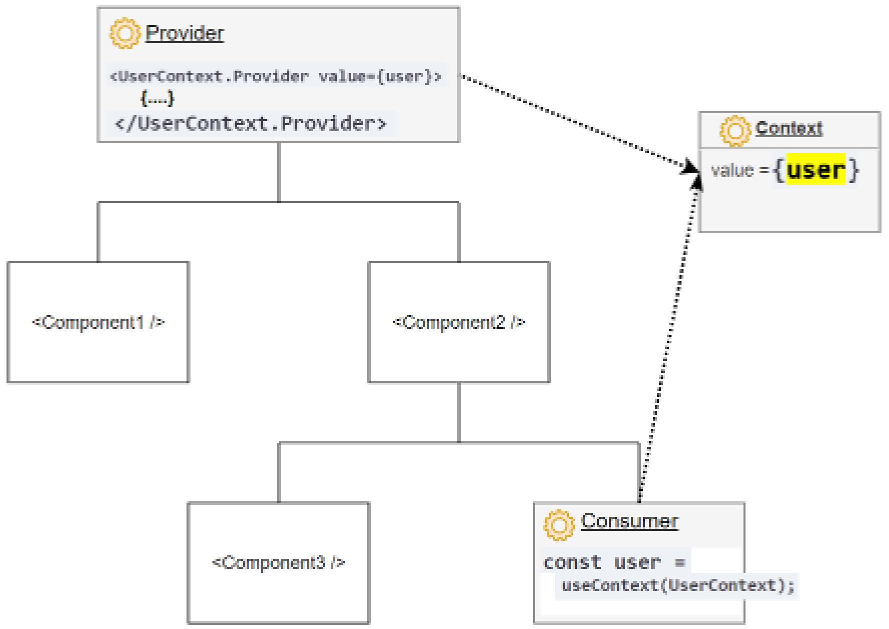
useRef vs. useState

- useState, useReducer, and useContext hooks triggering re-renders when a state variable changes
- useRef remembers the state value but change of value does not trigger rerender
  - The values of refs persist (specifically the current property) throughout render cycles

useContext

- Share state between deeply nested components more easily "prop drilling" (i.e., pass the state as "props" through each nested component)
- Using the context requires 3 steps: creating, providing, and consuming the context





useContext – Define global variables and functions

1. **Create a context** (i.e., a global container to provide global variables and functions available to all components)

```
import React from 'react';
const UserContext = React.createContext();
export default UserContext;
```

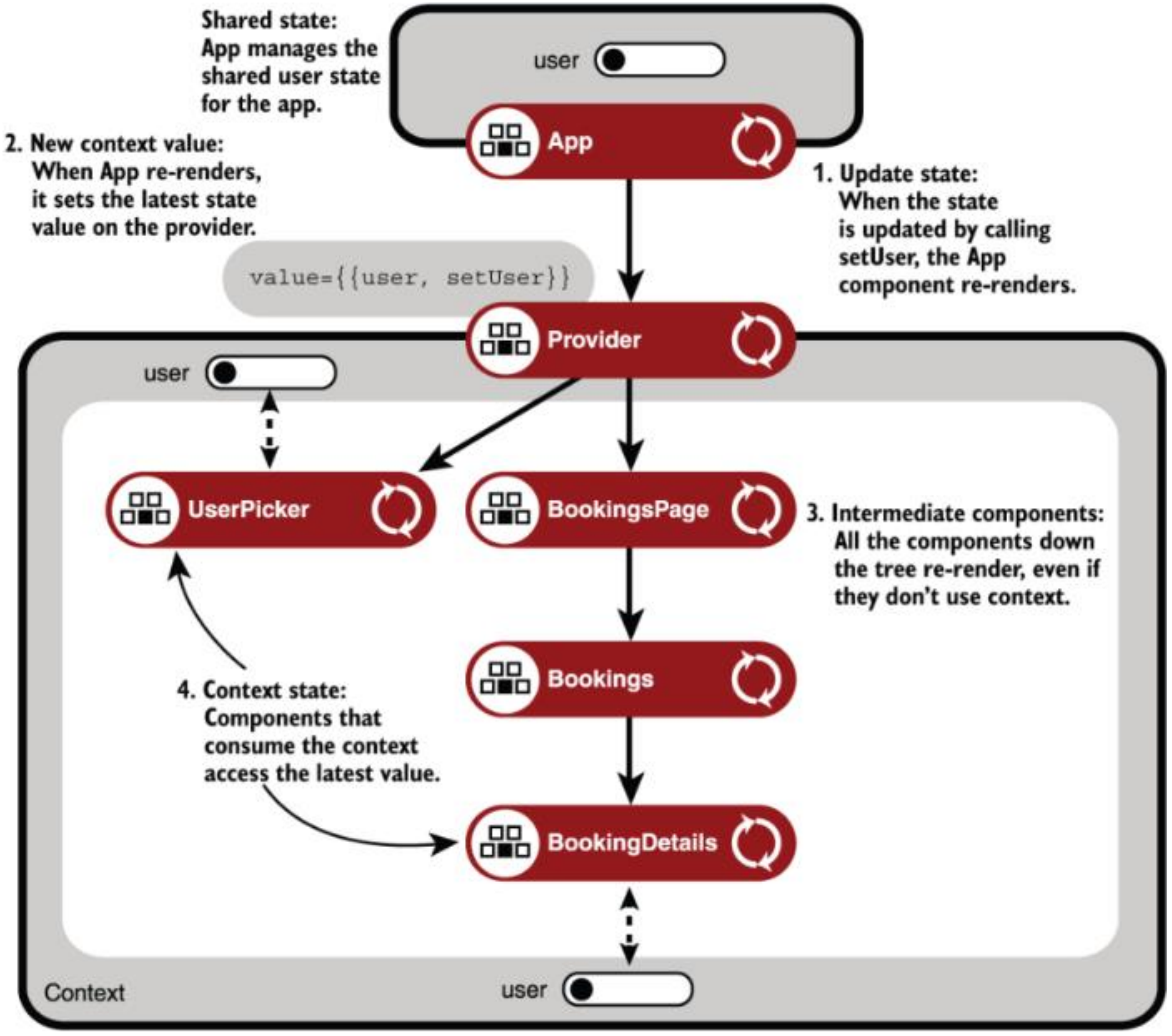
2. **Provider places global variables / functions in the context**

```
import UserContext from './components/UserContext';
function App() {
   return (
     <UserContext.Provider value={ user }>
            <Welcome /> …
      </UserContext.Provider>
   );
}
```

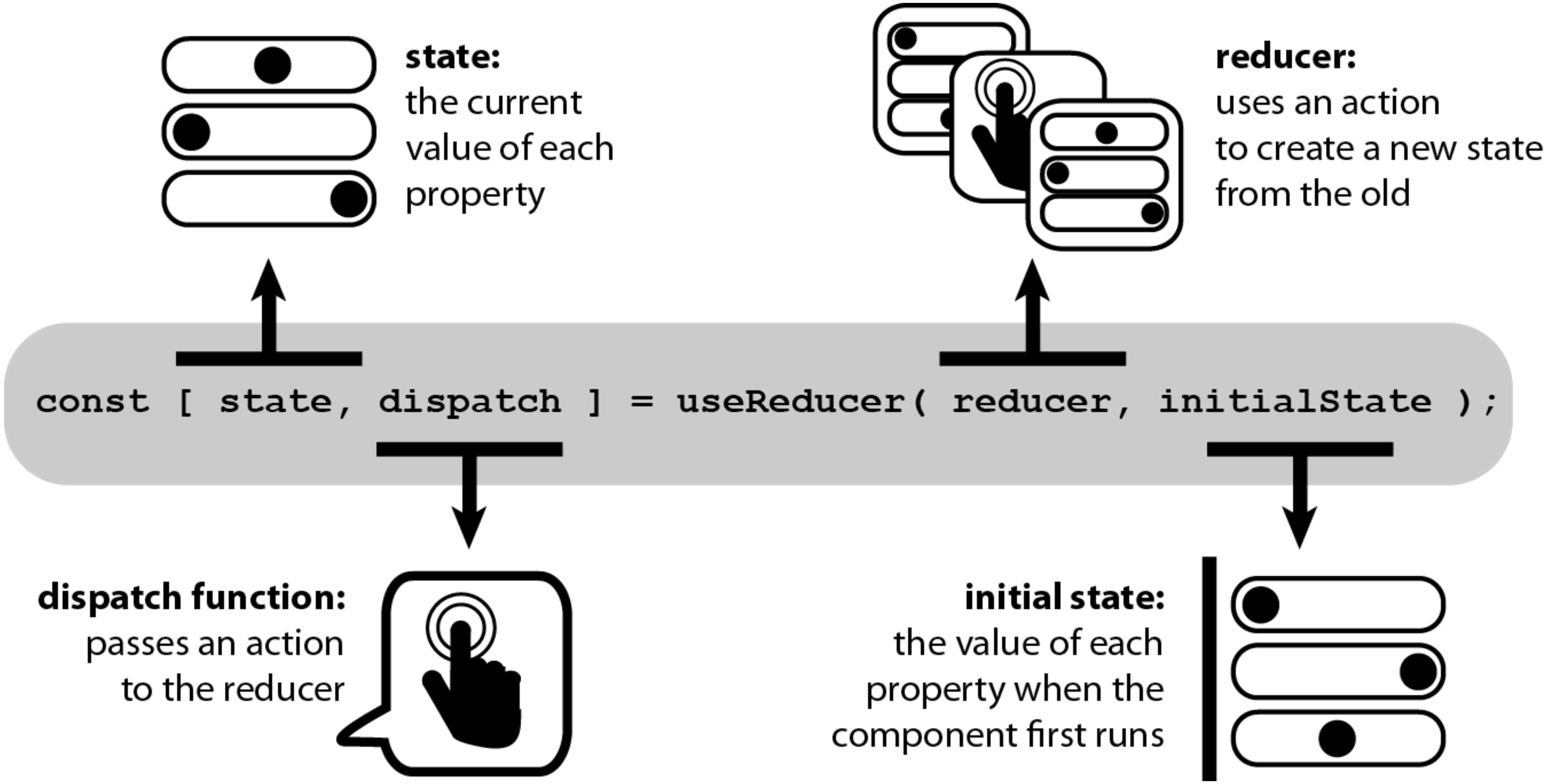
3. **Consumer access the global variables / functions in the context**

```
import React, {useContext} from "react";  import UserContext from './UserContext';
export default function Welcome() {
    const user = useContext(UserContext);
    return <div>You are login as: {user.username}</div>;
}
```

Shared State Example


Shared state:
App manages the shared user state for the app.
user
App
2. New context value:
When App re-renders, it sets the latest state value on the provider.
1. Update state:
When the state is updated by calling setUser, the App component re-renders.
value={{user, setUser}}
Provider
user
UserPicker
BookingsPage
3. Intermediate components:
All the components down the tree re-render, even if they don't use context.
Bookings
4. Context state:
Components that consume the context access the latest value.
BookingDetails
Context
user

useReducer: manage multiple related state variables



```
const [ state, dispatch ] = useReducer( reducer, initialState );
```

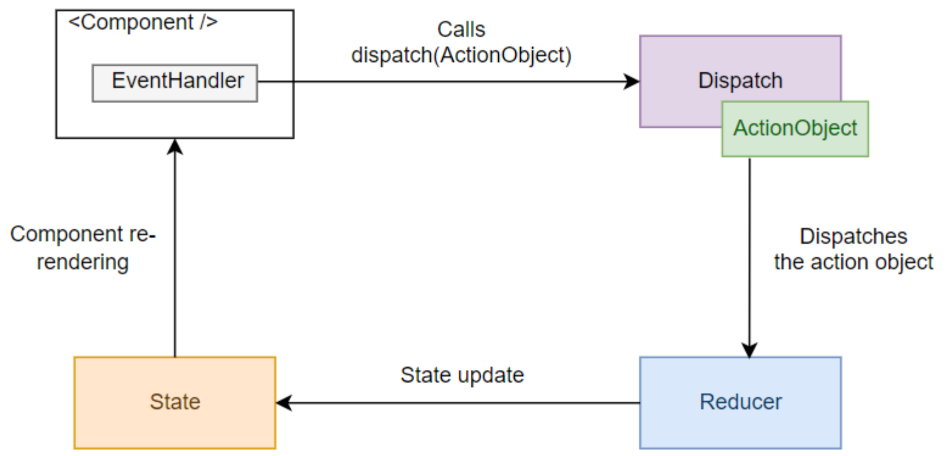
state:
the current
value of each
property

reducer:
uses an action
to create a new state
from the old

dispatch function:
passes an action
to the reducer

initial state:
the value of each
property when the
component first runs

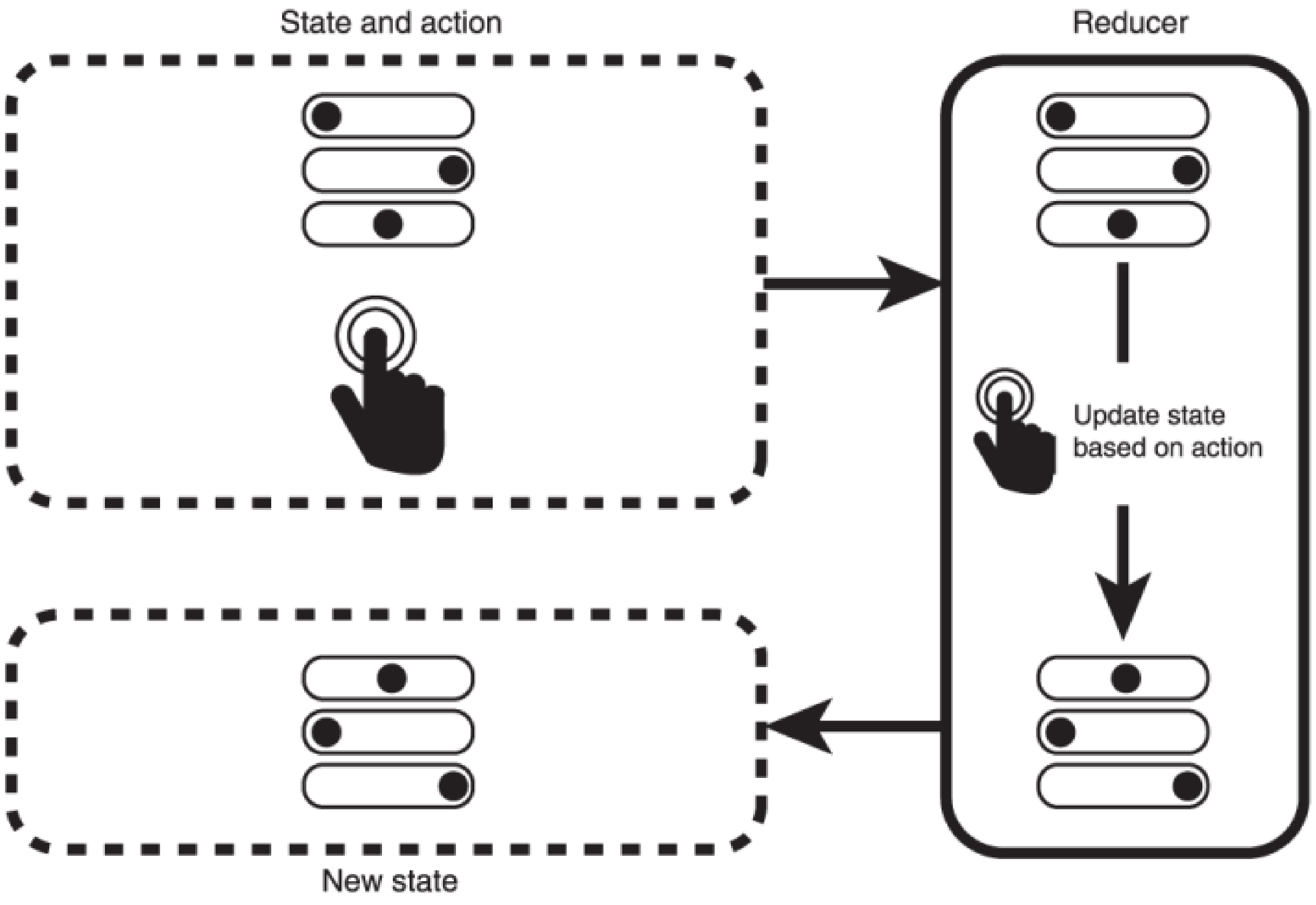
useReducer()



- As a result of an event handler, you call the dispatch function with the action object
- Then React redirects the action object and the current state value to the reducer function
- The reducer function uses the action object (and an optional payload) and performs a state update, returning the new state
- useReducer() returns the new state value: [newState, …] = useReducer(…)

If the state has been updated, React re-renders the component

A reducer takes a state and an action and returns a new state



Summary

- Hooks are functions which "hook into" React state and lifecycle features from function components
- **useState** : manage state
- **useEffect**: perform side effects and hook into moments in the component’s life cycle
- **useContext:** share data with child components without prop drilling using
- **useRef**: access DOM elements using
- **useReducer:** manage multiple related state variables

Resources

- **Hooks at a Glance**

https://reactjs.org/docs/hooks-overview.html

- React Hooks in Action textbook

https://learning.oreilly.com/library/view/react-hooks-in/9781617297632/

- Useful list of resources

https://github.com/rehooks/awesome-react-hooks