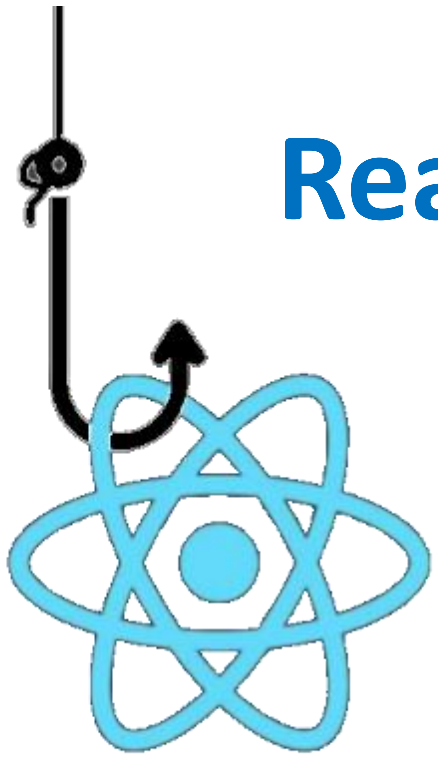
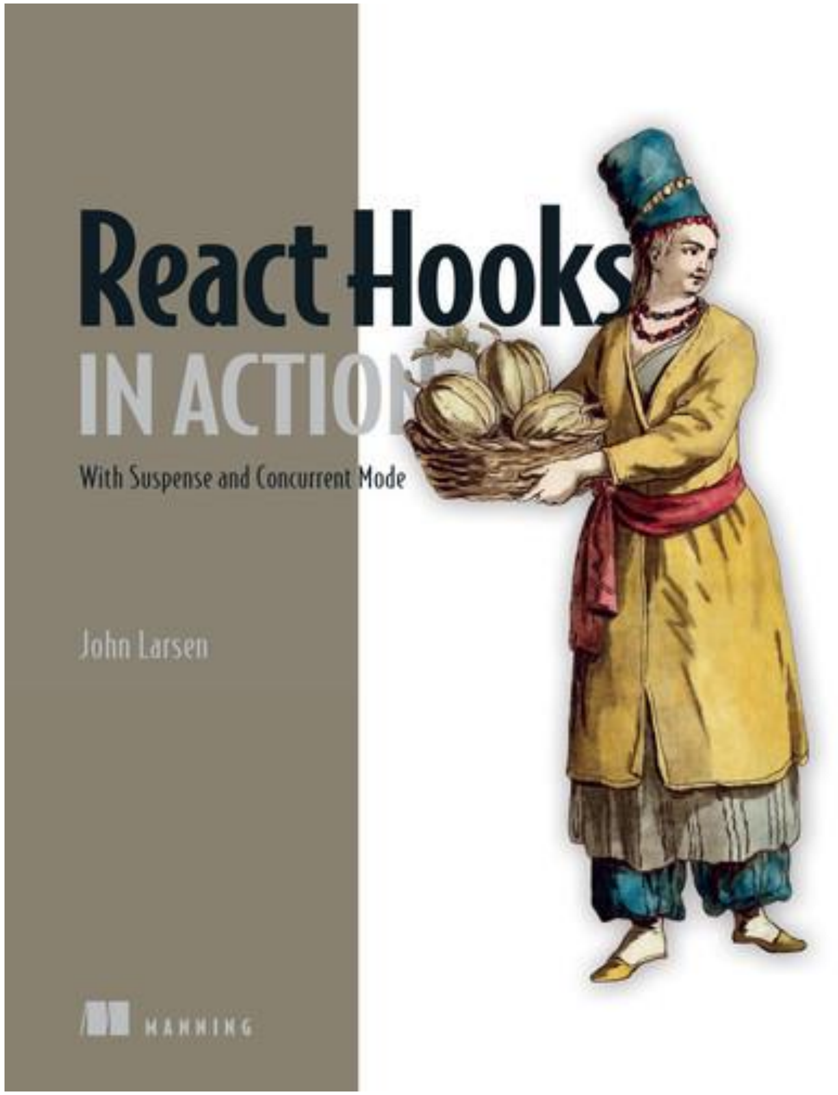


React Hooks

Outline

Slides are based on



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

What is Hook?

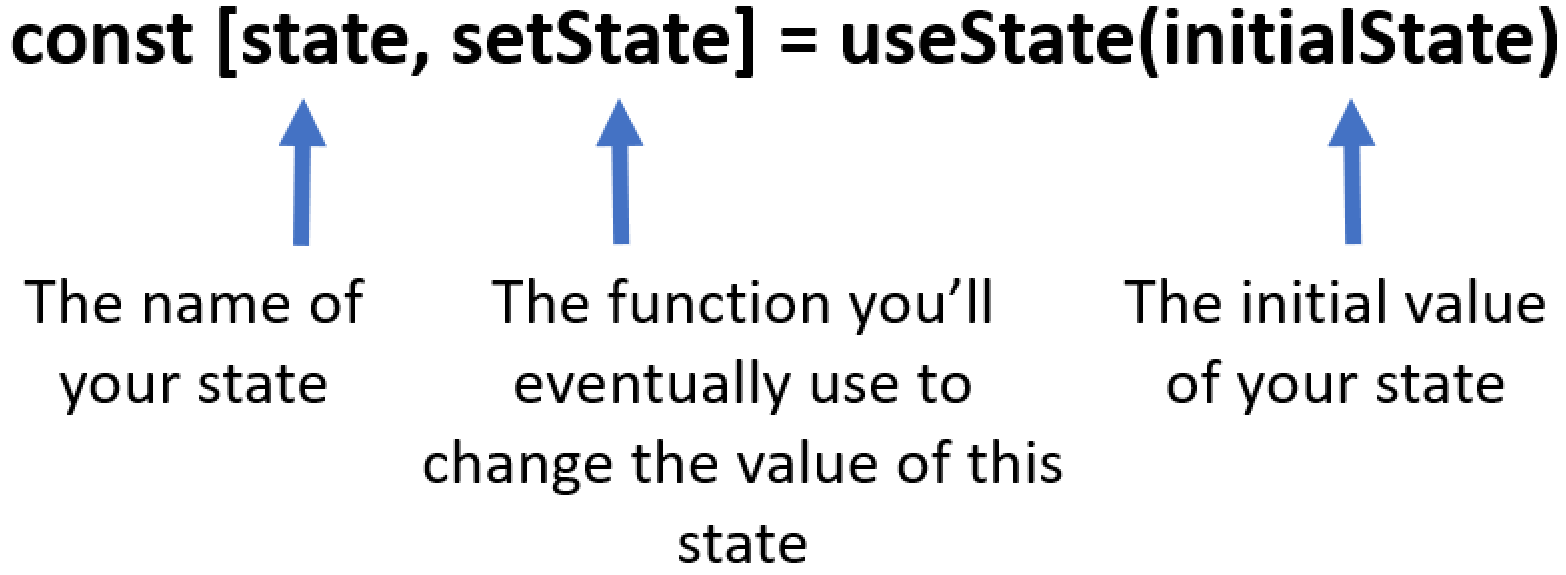
- A Hook is a special function that lets you **hook** into React features such as state and lifecycle methods
- There are 3 rules for hooks:
    - Hooks can only be called inside React function components.
    - Hooks can only be called at the top level of a component.
    - Hooks cannot be conditional

Common Hooks

Back

useState: creates a state variable

- Used for basic state management inside a component



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

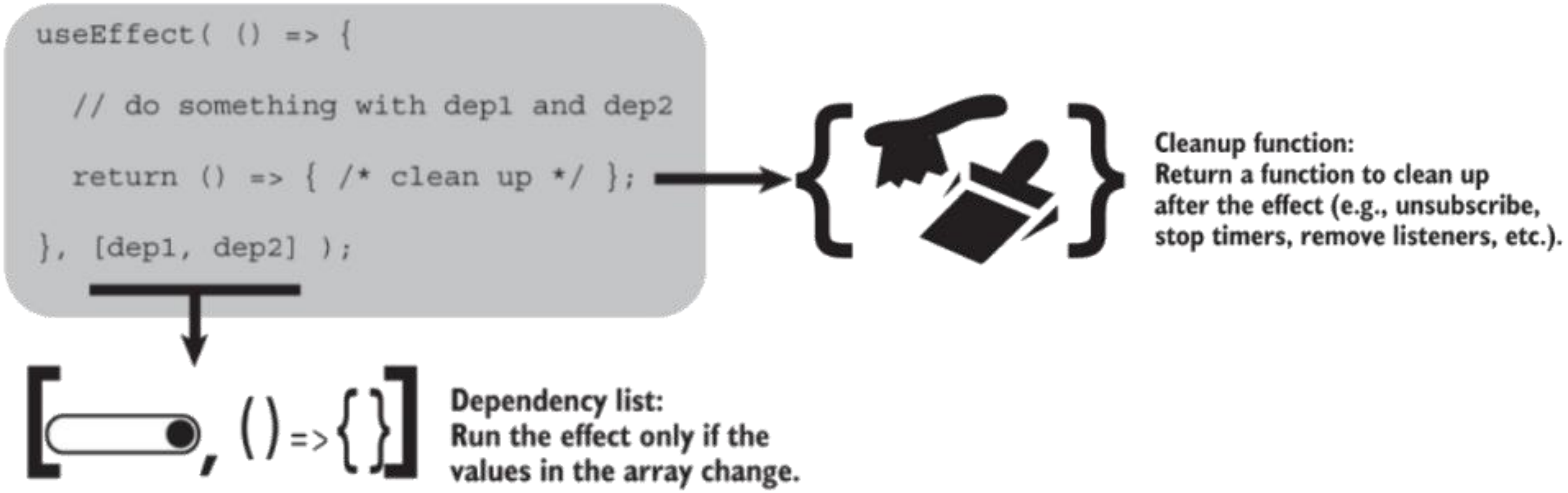
The name of your state

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] );
```

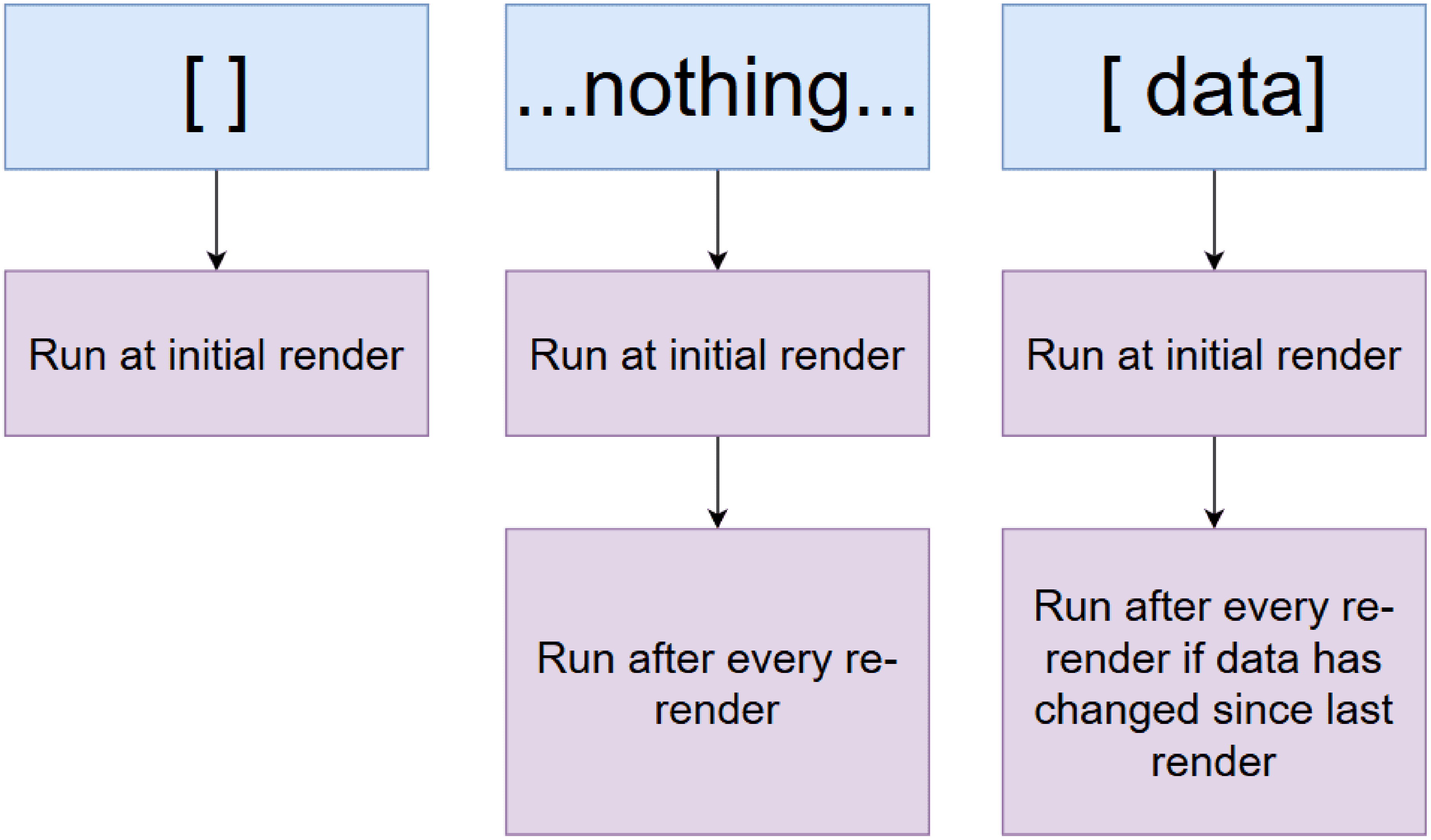
Cleanup function:
Return a function to clean up after the effect (e.g., unsubscribe, stop timers, remove listeners, etc.).
Dependency list:
Run the effect only if the values in the array change.

Common side effects

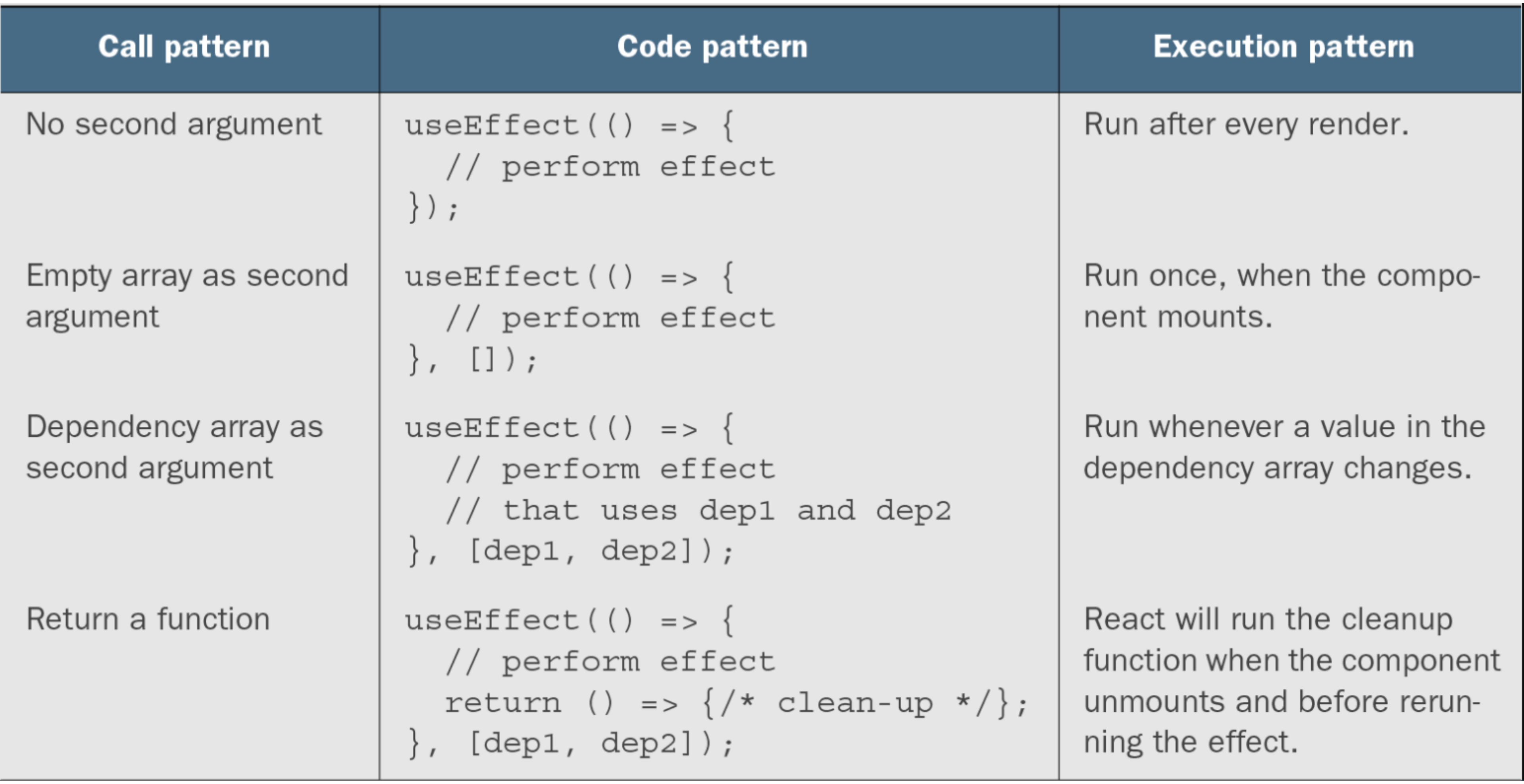
Common side effects include:

- 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.
```

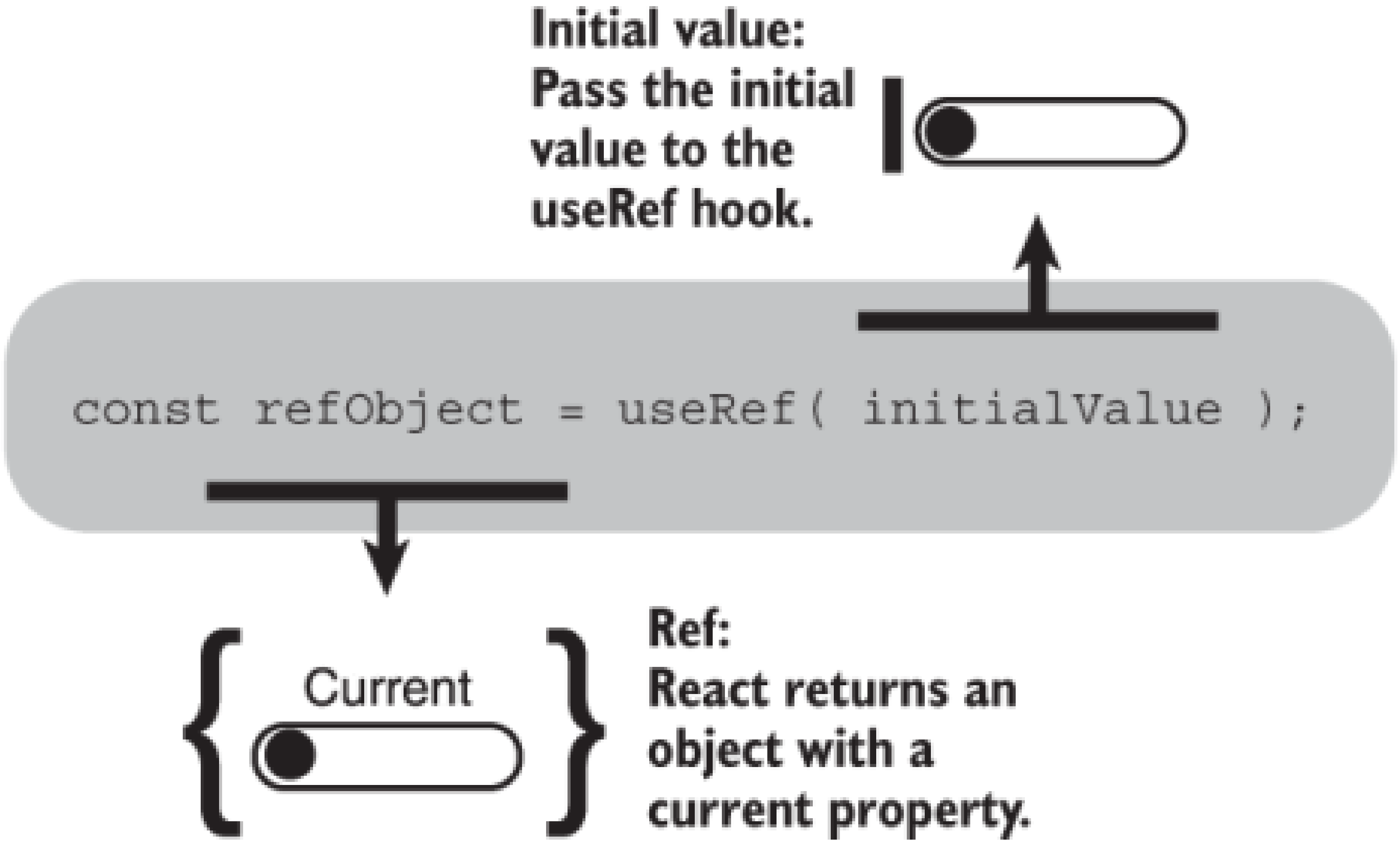
- **Executing a function every time a state variable changes**

```
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]);
```

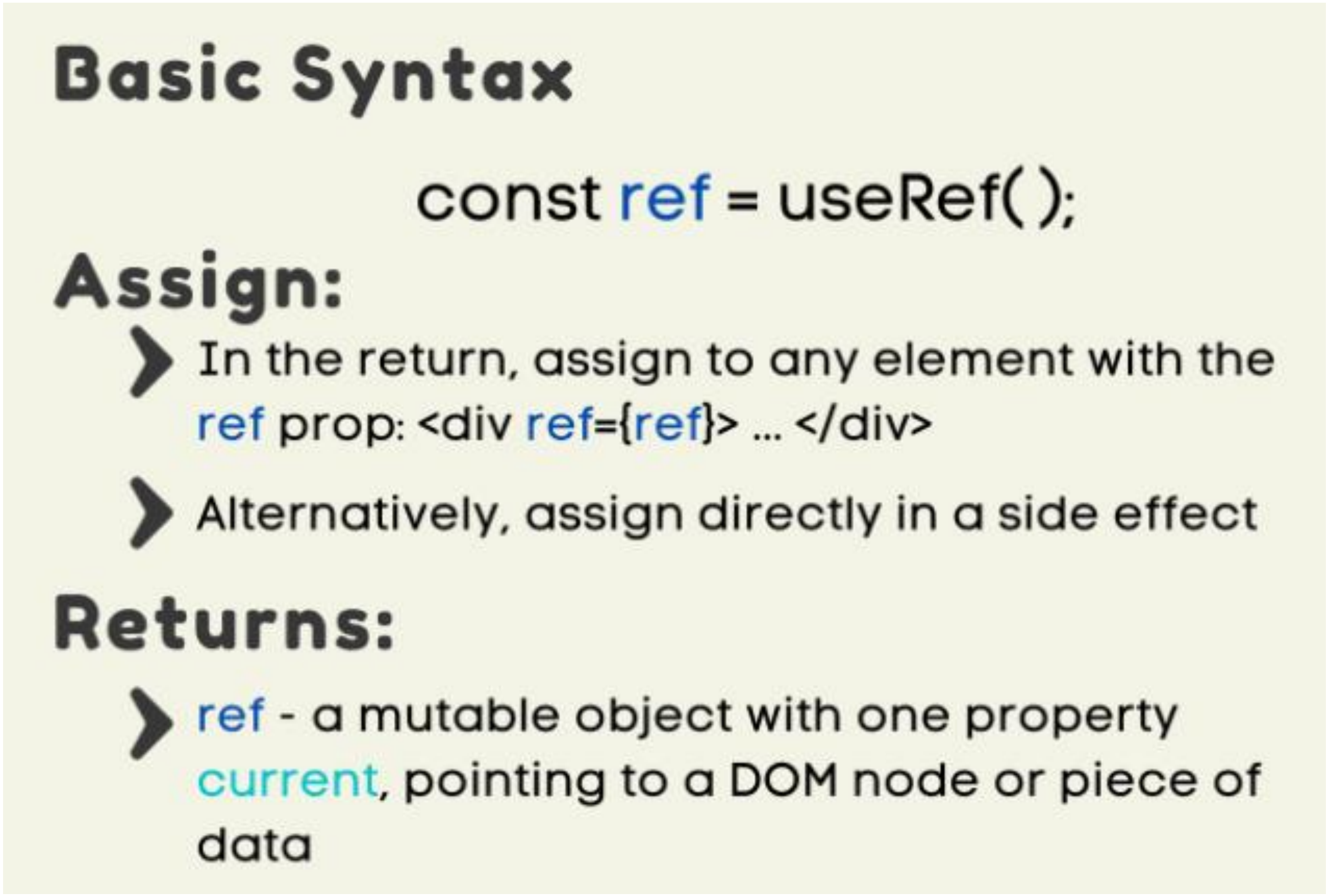
If 2nd parameter is not set, then the useEffect function will run on every re-render

useRef

- Allows updating state without causing a re-render
- Commonly used to access DOM elements


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 to access DOM elements



Basic Syntax

```
const ref = useRef( );
```

Assign:

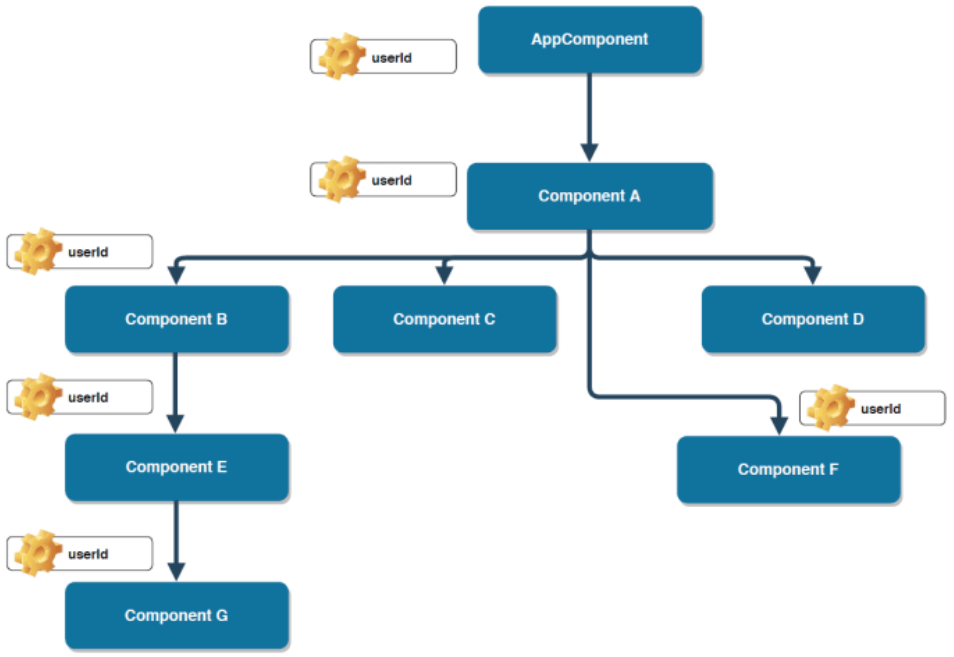
- In the return, assign to any element with the ref prop: `<div ref={ref}> ... </div>`
- Alternatively, assign directly in a side effect

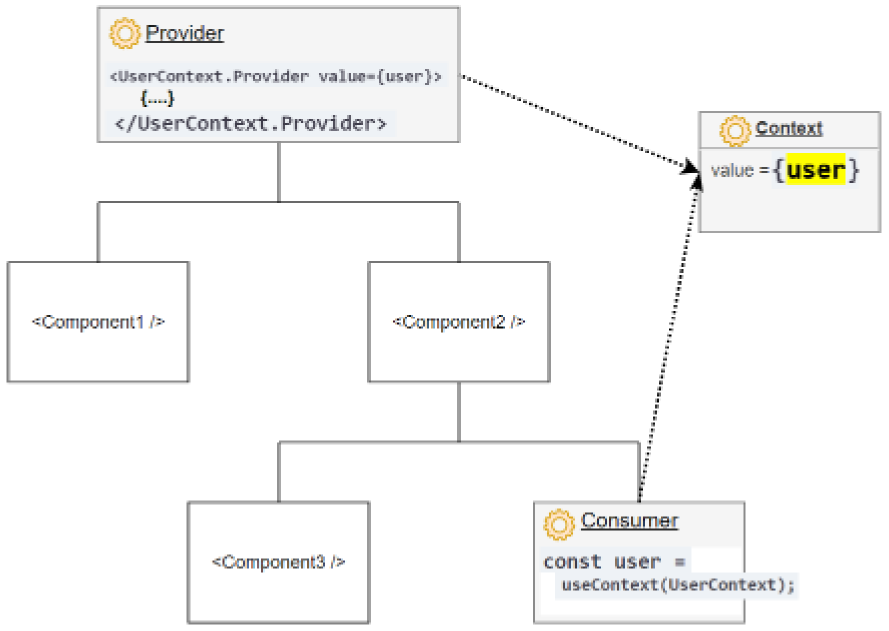
Returns:

- ref - a mutable object with one property current, pointing to a DOM node or piece of data

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





```
<UserContext.Provider value={user}>
   {....}
</UserContext.Provider>
```

```
const user =
  useContext(UserContext);
```

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;
```

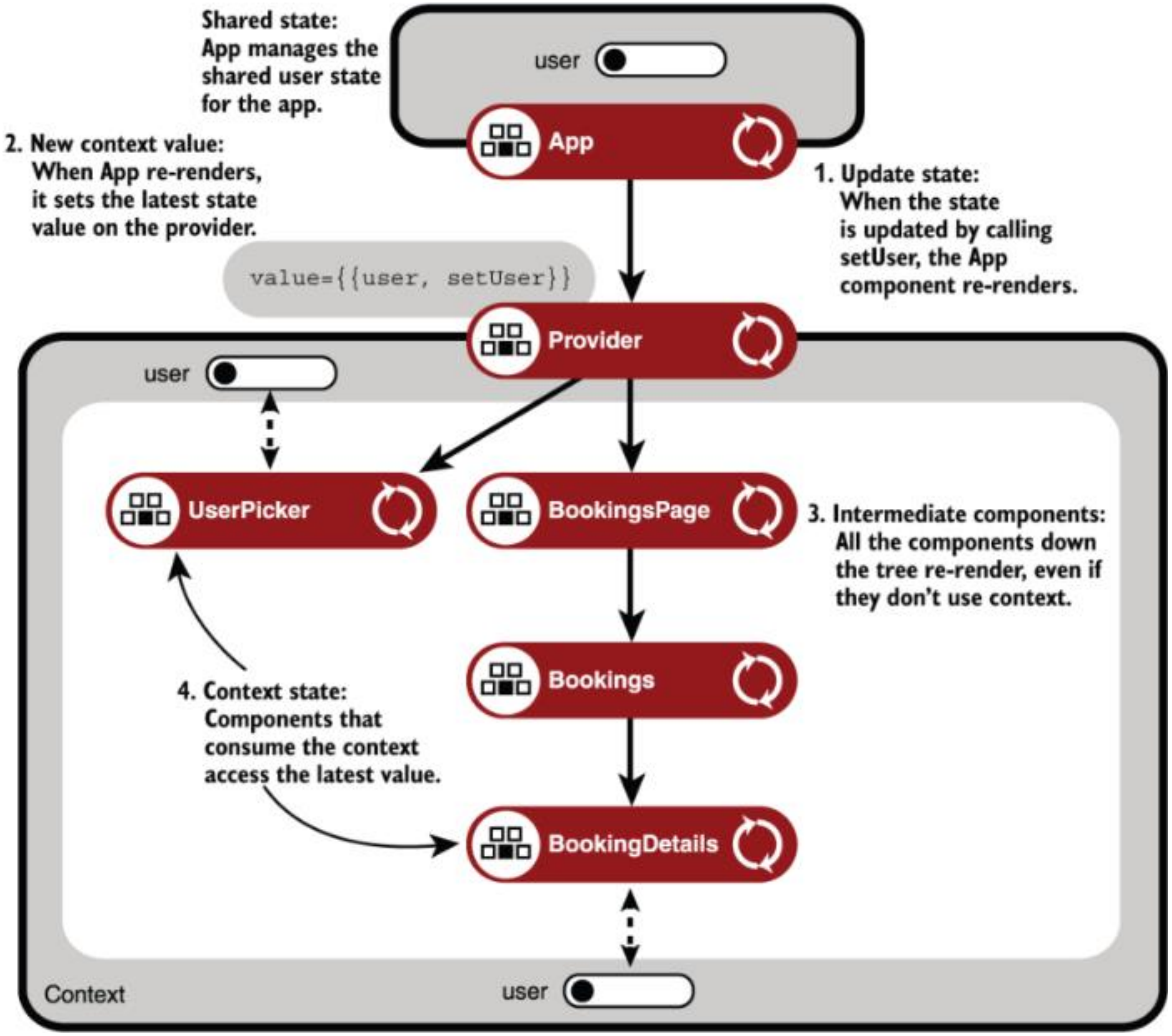
2. **Provider places global variables / functions in the context**

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

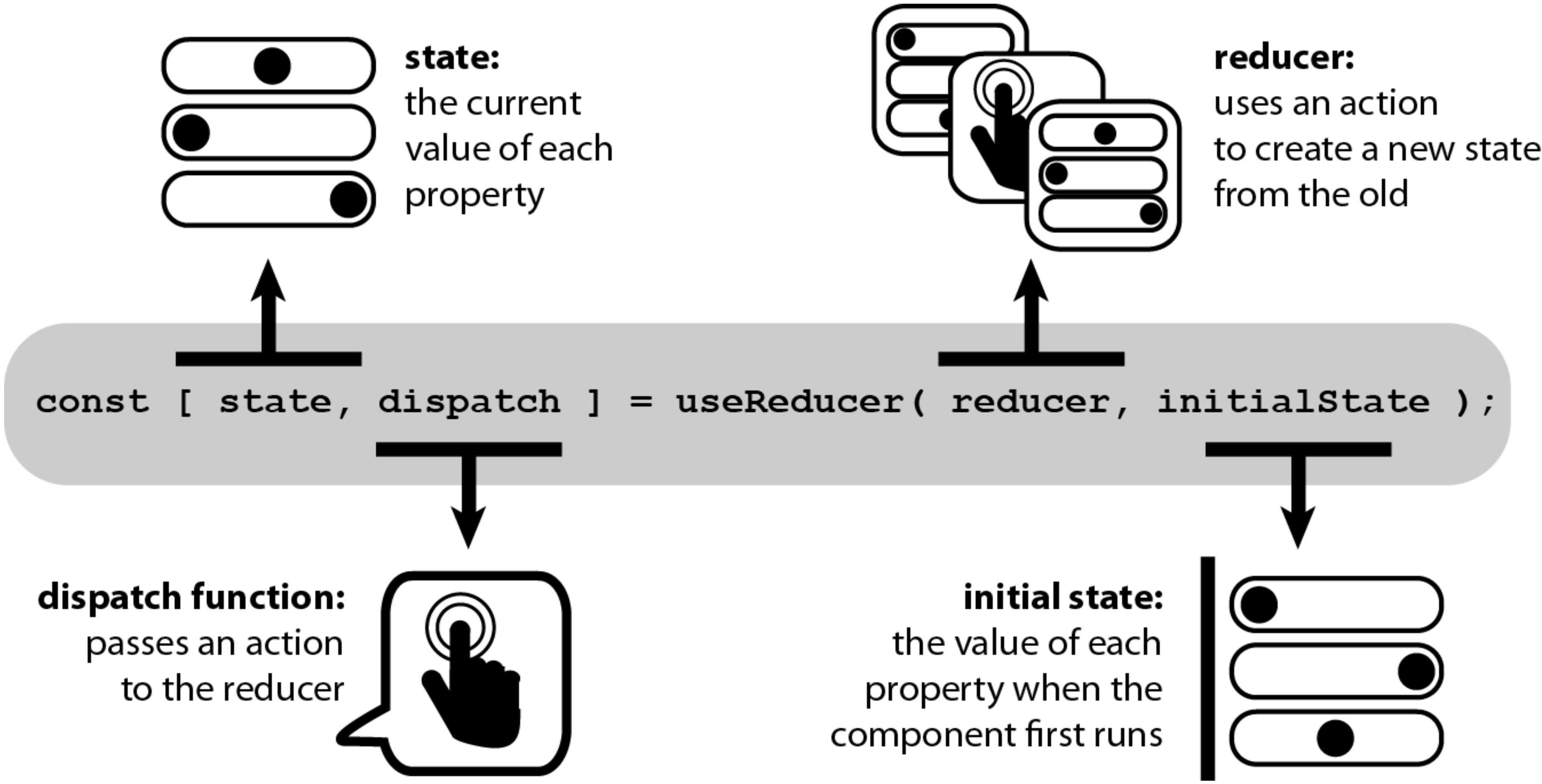
3. **Consumer access the global variables / functions in the context**

```
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.
value={{user, setUser}}
1. Update state:
When the state is updated by calling setUser, the App component re-renders.
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 );
```

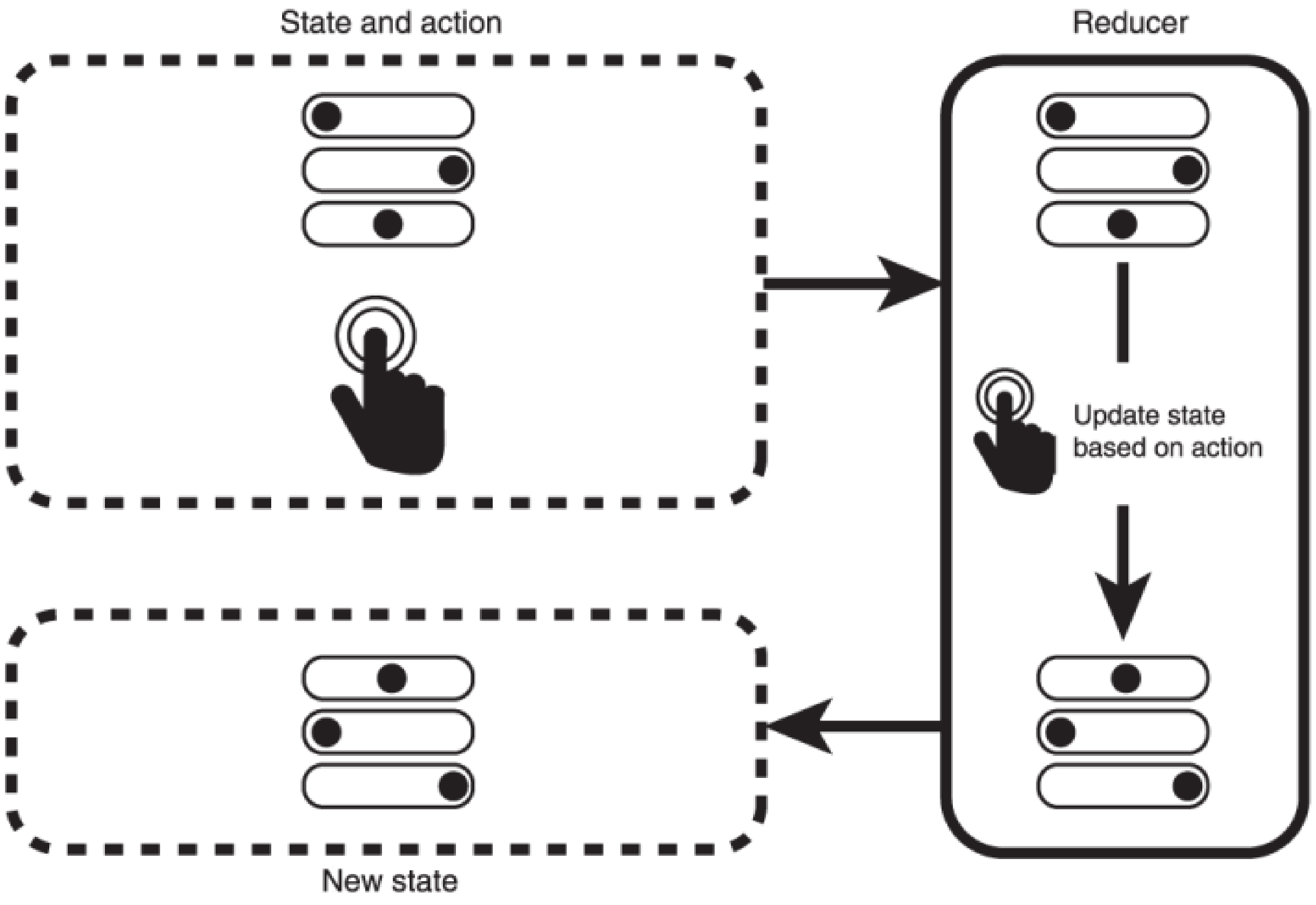
state:
the current
value of each
property

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

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


State and action
Reducer
Update state based on action
New state