React Course

Flavio Copes

Updated August 8, 2026

Contents

Learn React by building interfaces from components, JSX, props, state, hooks, events, forms, lists, and predictable one-way data flow.

What you'll learn: Build a small interactive React application with reusable components, local state, forms, lists, and one carefully chosen effect.

Start a React project

Understand React, create a project, and inspect its entry point.

What React is

Understand React as a library for describing user interfaces with components and updating the DOM from changing data.

React lets you describe an interface as a result of data.

```
function Greeting({ name }) {
  return <h1>Hello, {name}</h1>
}
```

The component does not create an `h1` with DOM commands. It returns a description of what should appear for the current `name`.

When props or state change, React follows the same cycle:

1. React calls the component again.
2. The component returns a new JSX snapshot.
3. React compares it with the previous result.
4. React updates the required DOM nodes.

Your component can stay simple because it answers one question: “What should the interface look like now?”

This is the core mental model for the course. You do not manually hide a message and enable a button. You update state, then render the message and button that match that state.

React is a library for the view layer. It gives you components, state, events, and a way to keep the

DOM in sync. A production application also needs decisions about routing, data loading, server rendering, authentication, and deployment. A framework can provide those pieces.

React is useful when an interface has stateful, reusable parts. A mostly static page may need only HTML and a little JavaScript. Do not add React merely because the page is on the Web.

Change the `name` prop in this example. Predict which DOM node needs to change, then confirm it in the Elements panel.

Create a React project with Vite

Scaffold a small local project for learning React fundamentals without relying on the deprecated Create React App tool.

Create a small Vite project so we can focus on React itself:

```
npm create vite@latest react-basics -- --template react
cd react-basics
npm install
npm run dev
```

Open the local URL printed by Vite. Keep the terminal running while you edit.

The generated project gives the browser normal HTML, JavaScript modules, and CSS. Vite transforms JSX during development and creates optimized assets for production. React still runs in the browser; Vite is the toolchain around it.

Delete the demo content from `src/App.jsx` and start with this:

```
export default function App() {
  return <h1>React basics</h1>
}
```

Save the file and confirm the page updates.

This setup is good for learning the library directly. For a production application, a framework can also decide how routing, data loading, server rendering, and deployment work. Choose that architecture from the product's needs rather than treating a bare Vite app as the answer to every React project.

If the page does not load, read the first terminal error before changing code. A missing dependency, wrong directory, and JSX syntax error need different fixes.

Follow the project entry point

Trace the HTML root element through createRoot to the first component rendered by React.

A Vite React project starts with ordinary HTML. `index.html` contains the DOM node React will manage:

```
<div id="root"></div>
<script type="module" src="/src/main.jsx"></script>
```

The JavaScript entry file finds that node and creates a React root:

```
import { createRoot } from 'react-dom/client'
import App from './App.jsx'
```

```
const container = document.getElementById('root')
const root = createRoot(container)

root.render(<App />)
```

`App` becomes the top component in the React tree. It can return HTML elements and other components, which return more elements and components.

Keep the two trees separate in your mind:

- The **component tree** describes which React components render other components.
- The **DOM tree** contains the browser elements React produced.

React DevTools shows the component tree. The Elements panel shows the DOM tree. A component may return several DOM elements, or return another component without adding a wrapper.

If `document.getElementById('root')` finds nothing, React has nowhere to render. Check that the HTML ID and JavaScript selector match.

Open both DevTools panels. Find `App` in one and its heading in the other.

Development and production builds

Use the fast development server while editing and create optimized static assets before deployment.

`npm run dev` starts the development server. It favors fast feedback, source maps, and useful warnings while you edit.

Development behavior is not the deployed application. The build may reveal an import that differs only by filename case, a missing environment value, or code that cannot be bundled.

Create the production assets with:

```
npm run build
```

Then inspect them locally:

```
npm run preview
```

The preview command serves the built files. It does not recreate your real hosting platform, so test routing, headers, and server features in the deployed environment too.

React can also do extra development work. Strict Mode may render components or rerun Effects to expose impure code and missing cleanup. Do not "fix" that by depending on production running fewer checks. Fix the component so repeating the setup is safe.

Before publishing, run the build, open the preview, check the browser console, and test one critical interaction. A page working in an already-open development tab is not enough evidence.

Install React Developer Tools

Inspect the component tree, props, state, and hooks instead of treating the rendered DOM as the complete application model.

React Developer Tools adds Components and Profiler panels to supported browser developer tools.

Use the Components panel to inspect React's model:

- which component rendered another component

- the current props
- state and Hook values
- which component owns changing data

Use the normal Elements panel to inspect the browser's model:

- actual DOM elements
- accessibility information
- matched CSS
- event and layout behavior

These views answer different questions. A wrong prop belongs in the component tree. A correct prop with broken spacing belongs in the DOM and CSS.

Select a component, change one state value through the interface, and watch the panel update. Then select the DOM element it produced.

The Profiler records which components rendered during an interaction. Use it after you observe a real performance problem. A render is not automatically expensive, and avoiding every render can make code harder to understand.

Do not edit props or state in DevTools and treat the result as a code fix. Use the inspection to find where the value originates, then change that source.

Components and JSX

Describe interfaces with reusable components and JavaScript markup.

Write your first component

Create a capitalized JavaScript function that returns one piece of the interface.

A component is a JavaScript function that returns a piece of the interface.

```
function Greeting() {
  return <h1>Hello</h1>
}
```

Use it from another component:

```
export default function App() {
  return (
    <main>
      <Greeting />
    </main>
  )
}
```

The capital letter matters. Lowercase JSX names such as `<main>` and `<h1>` mean built-in browser elements. A capitalized name such as `<Greeting>` refers to your JavaScript function.

React calls `Greeting()` while rendering. The returned JSX becomes part of the next interface snapshot.

Declare component functions at the top level of the module. Defining `Greeting` inside `App` creates a new component type on every render and can reset state below it.

A component should represent a meaningful UI responsibility. Do not wrap every `div` in a new function. Extract a component when the piece repeats, owns behavior, or becomes clearer with a name.

Render two `<Greeting />` instances. Confirm React calls the same component function for two different positions in the tree.

Compose components

Build a larger interface by nesting small components and keeping each one responsible for a coherent piece of UI.

Components become useful when they compose into a larger interface.

```
function Article() {
  return (
    <article>
      <h2>Forms are an HTTP interface</h2>
      <p>The browser sends named values to a URL.</p>
    </article>
  )
}

function Page() {
  return (
    <>
      <Header />
      <main>
        <Article />
      </main>
    </>
  )
}
```

`Page` controls the page structure. `Article` controls one content item. `Header` can be reused across pages.

Composition keeps data flow visible. A parent can pass data down through props and receive events through callback props. The child does not need to know the whole application.

Split a component when a piece has a clear responsibility, repeats, owns state, or is easier to test on its own. Keep one-off markup together when another file and prop interface would add more ceremony than clarity.

Do not call a component as `Article()`. Use `<Article />` so React can track its position, Hooks, and state as part of the tree.

Move the article text into props next. If the prop list becomes awkward, reconsider where the component boundary belongs.

JSX is markup inside JavaScript

Read JSX as a syntax transformed into React element descriptions rather than as a string or a second template language.

JSX is a syntax for writing element descriptions inside JavaScript.

```
const heading = <h1>Latest notes</h1>
```

This is not an HTML string. A build tool transforms it into JavaScript that describes the element type, properties, and children React should render.

JSX looks like HTML because React ultimately creates DOM elements, but it follows JavaScript module rules. Values come from variables and imports in the current file.

Because JSX is an expression, you can return it, assign it, pass it to a function, or choose it with a condition:

```
function Status({ saved }) {
  const message = saved
    ? <p>Changes saved.</p>
    : <p>You have unsaved changes.</p>

  return message
}
```

The browser does not execute JSX directly. Vite transforms it during development and build.

Do not build JSX by joining untrusted strings and inserting them as HTML. Normal text expressions are escaped by React. APIs that inject raw HTML bypass that protection and need a separate security review.

Open the built JavaScript or a JSX compiler tool and inspect one transformed element. You do not need to memorize the output; notice that JSX becomes JavaScript data, not an HTML file.

The basic JSX rules

Close tags, return one root, and use React property names where JSX differs from HTML.

JSX is stricter than HTML in a few useful ways. The strictness is not pedantry: JSX compiles to JavaScript, and JavaScript cannot tolerate the ambiguity browsers forgive in HTML. Learn these rules once and the error messages start making sense.

Every tag must close. HTML lets you write `<img>` or `<input>` bare; JSX does not:

```
<img src="/avatar.jpg" alt="Ada" />
<input name="email" type="email" />
```

The self-closing `/>` is mandatory for elements without children. Leave it off and the compiler stops with an unterminated-tag error instead of guessing what you meant.

A component returns one root value, because a `return` statement can only return one thing. Use a meaningful parent when the DOM needs one, or a fragment when it does not:

```
return (
  <>
    <Header />
    <main>{content}</main>
  </>
)
```

If you forget, the error reads *"Adjacent JSX elements must be wrapped in an enclosing tag"*. That

message means exactly this rule: two siblings at the top of a return need a shared parent.

Most properties use DOM-oriented JavaScript names. Use `className` for CSS classes and camelCase for events such as `onClick` and `onChange`. The names come from the DOM's JavaScript API rather than from HTML attribute spelling.

Accessibility and data attributes keep their HTML spelling:

```
<button aria-label="Close" data-action="dismiss">×</button>
```

Use `htmlFor` on a label because `for` has another meaning in JavaScript:

```
<label htmlFor="email">Email</label>
<input id="email" name="email" />
```

The realistic failure is quiet, not loud. Write `class` instead of `className` and the page still renders, but React logs a warning and your CSS may not apply. React reports many invalid properties in the console. Read the warning instead of guessing; it usually names the property and the correction.

Convert a small HTML form to JSX. Check closing tags, `className`, `htmlFor`, and the single returned root. Then inspect the output DOM to confirm it remains semantic HTML: the JSX rules are for the compiler, and the browser still receives ordinary elements.

Put JavaScript expressions in braces

Insert values and calculate attributes with expressions while keeping statements outside the returned JSX.

Curly braces switch from JSX markup to a JavaScript expression:

```
const name = 'Ada'

return <h1>Hello, {name.toUpperCase()}</h1>
```

You can use variables, property access, function calls, array methods, and conditional expressions.

Statements do not fit inside braces. An `if` statement changes control flow but does not produce a value. Put it before the return:

```
function Price({ amount }) {
  if (amount === 0) {
    return <strong>Free</strong>
  }

  return <span>€{amount}</span>
}
```

Use a ternary when both choices fit naturally inside the surrounding markup:

```
<p>{online ? 'Online' : 'Offline'}</p>
```

Booleans, `null`, and `undefined` render nothing. This makes `showDetails && <Details />` useful, but be careful with numbers. `count && <List />` renders `0` when count is zero. Write `count > 0 && <List />` when that is the real condition.

Keep expressions readable. Calculate a complex value before the JSX and give it a clear name.

Try rendering the values `0`, `false`, `null`, and an empty string. Inspect what appears in the DOM.

Set attributes and styles

Pass strings, expressions, booleans, and style objects through JSX properties.

A quoted JSX property is a string. Braces pass a JavaScript value:

```
<img
  src={avatarUrl}
  alt={name}
  width={160}
  height={160}
  hidden={!avatarUrl}
/>
```

Here `src` and `alt` receive strings, width and height receive numbers, and `hidden` receives a boolean.

Do not quote an expression:

```
<img src="{avatarUrl}" alt="Profile" />
```

That sends the literal text `{avatarUrl}` to the browser.

Use CSS classes for most styling:

```
<p className={saved ? 'status status--saved' : 'status'}>
  {saved ? 'Saved' : 'Not saved'}
</p>
```

Inline styles receive an object with camelCase properties:

```
<div style={{ width: `${progress}%` }} />
```

Inline styles are useful when a value genuinely comes from data. Classes are usually clearer for hover, focus, media queries, and a shared visual system.

React does not make inaccessible attributes safe. An image still needs useful alt text, and a clickable `div` is still the wrong control. Prefer semantic elements before adding behavior and styles.

Change `saved` and `progress`, then inspect the actual class and style attributes in the DOM.

Keep rendering pure

Make component output depend only on props, state, and context without changing outside values during render.

Rendering must be a calculation, not an operation on the outside world.

Given the same props, state, and context, a component should return the same JSX. This lets React pause, restart, or repeat rendering safely.

This component is impure:

```
let nextId = 0

function Task() {
  nextId += 1
  return <p>Task {nextId}</p>
}
```

Every render changes a value outside the component. Rendering twice produces different output, and separate component instances affect each other.

Do not mutate props, write storage, start timers, send requests, or change the DOM while rendering.

Put work where its cause belongs:

- A click-caused request belongs in the click handler.
- A value derived from props belongs in the render calculation.
- Synchronization with an external system belongs in an Effect.

Local mutation is fine when the value was created during this render:

```
const rows = []

for (const task of tasks) {
  rows.push(<li key={task.id}>{task.title}</li>)
}
```

The new `rows` array does not affect an earlier render or another component.

Strict Mode intentionally repeats some development work to reveal impurities. If a repeated render creates duplicate data or requests, fix where that side effect occurs rather than disabling the check.

Add a `console.log` to a component and observe renders. Then verify that repeating the component call does not change application data.

Check your understanding: components and JSX

Check component naming, JSX syntax, expressions, property names, roots, fragments, and rendering purity.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Props and state

Pass data down and remember information between renders.

Pass data with props

Give a child component data through JSX attributes and read those values from the component parameter.

Props are the inputs to a component. The parent chooses them for the current render.

```
function Greeting({ name, unreadCount }) {
  return (
    <p>
      Hello, {name}. You have {unreadCount} unread messages.
    </p>
  )
}
```

```
export default function App() {
  return <Greeting name="Ada" unreadCount={3} />
}
```

Quoted values are strings. Braces pass JavaScript values, so `unreadCount={3}` passes a number.

When `App` renders again with different props, React calls `Greeting` again. The child receives a new snapshot of those values and returns matching JSX.

Props can contain strings, numbers, objects, arrays, functions, and JSX. Keep the interface focused. If a component needs many unrelated props, it may have too many responsibilities.

The child should not need to know where the data came from. It might come from state, a route, or a server response. It only needs the contract represented by its props.

Render two greetings with different values. Then deliberately pass `unreadCount="3"` and inspect the type. JSX does not infer a number from a quoted attribute.

Treat props as read-only

Ask the parent to provide a new value instead of mutating an input inside the child.

Props are read-only snapshots. A child must not assign to them or mutate an object received through them.

This breaks one-way data flow:

```
function Task({ task }) {
  task.done = true
  return <p>{task.title}</p>
}
```

The object belongs to the parent. Mutating it changes data from an earlier render without asking the owner to update.

When the child needs a change, pass an event callback:

```
function Task({ task, onToggle }) {
  return (
    <label>
      <input
        type="checkbox"
        checked={task.done}
        onChange={() => onToggle(task.id)}
      />
      {task.title}
    </label>
  )
}
```

The child reports the event. The parent updates its state and passes a new `task` prop on the next render.

Avoid copying a prop into state by default:

```
const [name, setName] = useState(user.name)
```

Later prop changes will not automatically replace that state. This is correct only when `name` is an intentional editable draft. Otherwise calculate from the prop directly.

Freeze a prop object during development and try the mutation example. The failure makes the ownership mistake easier to see.

Compose wrappers with children

Render the nested content a parent places between a component opening and closing tag.

Nested JSX arrives in the special `children` prop.

```
function Panel({ title, children }) {
  return (
    <section className="panel">
      <h2>{title}</h2>
      {children}
    </section>
  )
}
```

The parent supplies the content:

```
<Panel title="Account">
  <p>Signed in as ada@example.com</p>
  <button>Sign out</button>
</Panel>
```

`Panel` controls the shared structure. It does not need a separate prop for every paragraph or button that might appear inside it.

This is composition. It is often clearer than adding configuration props such as `showButton`, `buttonText`, and `bodyType` for every variation.

Children are still values from the parent render. The wrapper can place them, hide them, or surround them with markup, but it should not assume a specific child structure unless that is part of a documented component contract.

Use semantic wrappers. A panel that represents an independent topic may be a `section`; a purely visual wrapper may be a `div`.

Create a second `Panel` containing a form. Notice that the wrapper stays reusable without knowing form details.

Add state with useState

Remember a value between renders and request a new render through its setter.

State lets a component remember information between renders.

```
import { useState } from 'react'

function Counter() {
  const [count, setCount] = useState(0)

  return (
```

```
    <button onClick={() => setCount(count + 1)}>
      Count: {count}
    </button>
  )
}
```

`useState(0)` gives this component position an initial value. During a render, `count` is that render's snapshot. Calling `setCount()` requests another render with a new value.

The setter does not change the existing `count` variable. React calls the component again and gives the next render its own value.

Use state when information changes over time and affects the rendered result. Keep derived values as calculations:

```
const completed = tasks.filter(task => task.done).length
```

Storing `completed` separately would create two values that can disagree.

State belongs to a component position. Rendering two counters creates two independent state values even though both call the same component function.

Click one of two counters and confirm only that instance changes. Then log `count` immediately after `setCount()` and explain why it still contains the current render's value.

State is a snapshot

Understand why a state variable does not change inside the event handler that has already captured the current render.

Every render receives a snapshot of props and state. The event handlers created during that render keep seeing that snapshot.

Consider three updates in one click:

```
function Counter() {
  const [count, setCount] = useState(0)

  function addThree() {
    setCount(count + 1)
    setCount(count + 1)
    setCount(count + 1)
  }

  return <button onClick={addThree}>{count}</button>
}
```

If `count` is `0`, every line requests `setCount(0 + 1)`. The next render shows `1`, not `3`.

When the next value depends on a queued previous value, pass an updater function:

```
setCount(current => current + 1)
setCount(current => current + 1)
setCount(current => current + 1)
```

React processes the queue in order: `0 → 1 → 2 → 3`.

Snapshots also explain delayed handlers:

```
function showLater() {
  setTimeout(() => alert(count), 2000)
}
```

The alert sees the `count` from the render that created `showLater`, even if another render happens before the timer runs.

This behavior prevents a running handler from changing underneath you. Use an updater when calculating the next state from previous state. Use a ref only when an asynchronous callback genuinely needs the latest mutable value without rendering it.

Try both `addThree` versions and predict the result before clicking.

Update objects and arrays without mutation

Create a new value for state so React receives a changed reference and previous render snapshots remain reliable.

Treat objects and arrays in state as immutable snapshots. Create the next value instead of changing the current one.

This mutation is a bug:

```
tasks[0].done = true
setTasks(tasks)
```

The array reference has not changed, and an object used by an earlier render was modified in place.

Use non-mutating array methods and object spread:

```
setTasks(currentTasks =>
  currentTasks.map(task =>
    task.id === id
      ? { ...task, done: true }
      : task
  )
)
```

The new array contains a new object for the changed task. Unchanged task objects keep their existing references.

Spread is shallow. For nested data, copy every changed level:

```
setUser(user => ({
  ...user,
  address: {
    ...user.address,
    city: 'Copenhagen'
  }
}))
```

Keep state shapes simple. Deeply nested state makes updates harder and increases the chance of accidental mutation.

For arrays, use `map()` to replace, `filter()` to remove, and spread to add. Avoid mutating methods

such as `push()`, `pop()`, and `splice()` on state arrays.

Log the old and new references with `===`. The array and changed object should differ; unchanged objects can remain equal.

State belongs to a position in the tree

Recognize when React preserves component state and use identity or a key when the interface needs a deliberate reset.

State is not stored inside a JSX tag. React associates it with a component type at a position in the rendered tree.

If the same component stays in the same position, React preserves its state. Changing its props does not reset it.

This matters for an editable chat draft:

```
<Chat contact={selectedContact} />
```

Switching contacts changes the prop, but `Chat` remains in the same tree position. Its input state stays. That could send one person's draft to another person.

Give each independent draft an identity:

```
<Chat key={selectedContact.id} contact={selectedContact} />
```

When the key changes, React removes the old component state and creates a new subtree.

Keys are not only for lists. They tell React which identity occupies a position among siblings. Use a key to reset state when the product meaning changes, not as a general way to force rendering.

Changing the component type also resets state. This is one reason not to define component functions inside another component: every render creates a new function identity.

Decide whether state should survive before adding a key. A tab switch may preserve an unfinished form. A recipient switch may need a clean draft.

Build a small input that switches between two contacts. Test it with and without the key and choose the behavior deliberately.

Check your understanding: props and state

Check read-only props, children, state updates, snapshots, functional setters, immutability, and re-rendering.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Hooks and effects

Use Hooks consistently and synchronize with external systems.

What Hooks are

Use React functions that connect a component to state, context, refs, and synchronization features.

Hooks are functions that let a component use React features such as state, context, refs, and Effects.

```
import { useState } from 'react'

function Details() {
  const [open, setOpen] = useState(false)
  // ...
}
```

Built-in Hooks include `useState`, `useEffect`, `useRef`, and `useContext`. Custom Hook names also start with `use`.

A Hook call belongs to the component position currently rendering. React relies on the call order to match each Hook with its stored value.

Think of Hooks as connections, not general utility functions:

- `useState` connects the component to remembered state.
- `useContext` reads a value from the nearest provider.
- `useRef` keeps a mutable value or DOM reference between renders.
- `useEffect` synchronizes with an external system after rendering.

A custom Hook packages repeated stateful behavior. It shares logic, not one state instance. Calling `useOnlineStatus()` in two components gives each call its own Hook state unless both subscribe to the same external source.

Do not add a Hook because a function starts with `use`. The name is a promise that the function follows Hook rules and may call other Hooks.

Open React DevTools and inspect a component with two `useState` calls. Change one value and notice how React preserves each Hook by position.

Follow the Rules of Hooks

Call Hooks at the top level of React components and custom Hooks so the call order stays stable between renders.

Call Hooks at the top level of a React component or custom Hook.

Do not call a Hook inside a condition:

```
if (isLoggedIn) {
  const [profile, setProfile] = useState(null)
}
```

React matches Hook state by call order. If one render calls the Hook and the next skips it, every later Hook can be matched with the wrong stored value.

Call the Hook unconditionally, then branch with the result:

```
const [profile, setProfile] = useState(null)

if (!isLoggedIn) {
  return <Login />
}
```

Do not call Hooks in loops, nested callbacks, event handlers, or after an early return. Call them

only while React is rendering a component or another Hook.

Put a condition inside an Effect when the synchronization itself is conditional. Split the component when two branches genuinely need different stateful structures.

Some data Hooks accept a value that pauses their work. You still call the Hook on every render, but move the condition into its argument. SWR, for example, skips a request when its key is `null`:

```
const { data } = useSWR(isLoggedIn ? '/api/user' : null, fetcher)
```

Use the React Hooks linter. It can catch invalid call positions and missing Effect dependencies before they become timing bugs.

Move one Hook below an early return and run the linter. Then restore it to the top level and explain why the order is stable again.

Effects synchronize external systems

Use an Effect when rendering must start or update a connection to something React does not control.

An Effect synchronizes the rendered React state with a system React does not control.

Examples include:

- a browser media API
- a network connection
- a map or chart library
- an event subscription
- a timer that must follow component visibility

Ask one question before writing `useEffect`: **Which external system needs to match which React values?**

If there is no external system, you probably do not need an Effect.

Do not use an Effect to derive display data:

```
// Avoid
useEffect(() => {
  setVisibleTasks(tasks.filter(task => !task.done))
}, [tasks])
```

Calculate it during rendering:

```
const visibleTasks = tasks.filter(task => !task.done)
```

The second version has one source of truth and avoids an extra render.

An action caused by a specific click also belongs in that event handler. Buying a product or submitting a form should not wait for an Effect to notice a state flag.

Effects are caused by rendering itself. A chat connection should exist whenever the chat room is rendered, regardless of which navigation event made it appear.

Review every Effect in a small component. Name the external system. If you cannot name one, try replacing the Effect with a render calculation or event handler.

Write an Effect and its dependencies

Declare synchronization after rendering and list every reactive value the Effect reads.

An Effect runs after React commits the updated interface.

```
import { useEffect } from 'react'

function TaskPage({ tasks }) {
  useEffect(() => {
    document.title = `Tasks: ${tasks.length}`
  }, [tasks.length])

  return <TaskList tasks={tasks} />
}
```

This Effect keeps the browser document title synchronized with the rendered task count.

The dependency list describes the reactive values the Effect reads. It is not a performance preference. When `tasks.length` changes, React runs the synchronization again.

React compares dependencies with their values from the previous render. An object or function created during every render is a new reference and can make an Effect rerun. Move unnecessary object creation inside the Effect or simplify the dependency rather than hiding it from the linter.

Do not suppress a dependency warning to force "run once" behavior. The warning often means the Effect reads a value that can become stale.

An empty array means the Effect does not read changing props or state. It does not mean "ignore values I used."

Change a task title without changing the list length, then add a task. Predict which action updates the document title and confirm it in the browser tab.

Clean up an Effect

Undo subscriptions, timers, connections, and stale asynchronous work before resynchronizing or removing the component.

An Effect that starts synchronization often needs to stop it.

Return a cleanup function:

```
useEffect(() => {
  window.addEventListener('online', handleOnline)

  return () => {
    window.removeEventListener('online', handleOnline)
  }
}, [])
```

React calls cleanup before this Effect synchronizes again with changed dependencies. It also calls cleanup when the component leaves the tree.

Think in pairs:

- subscribe → unsubscribe

- connect → disconnect
- start timer → clear timer
- add listener → remove listener
- start request → ignore or abort stale result

Without cleanup, remounting the component can add another listener. One browser event then runs the handler several times.

The cleanup must refer to the same subscription or handler that setup created. An anonymous function passed separately to `removeEventListener()` is a different function and does not remove the listener.

Cleanup should undo the synchronization from that Effect, not reset unrelated application state.

Mount and unmount the component three times, then dispatch one event. The handler should run once. Strict Mode can help reveal this leak during development.

Why Effects run twice in development

Treat development remounting as a test that exposes missing cleanup instead of adding a flag to hide the second call.

Strict Mode adds development checks that expose unsafe rendering and missing Effect cleanup.

For an Effect, React may run this sequence in development:

1. Start synchronization.
2. Clean it up.
3. Start it again.

This simulates leaving and returning to the screen. A correct Effect remains safe because cleanup fully stops the first synchronization.

Do not hide the second run with a ref flag:

```
if (hasRun.current) return
hasRun.current = true
```

That can hide the warning while leaving the real remount bug. If the user navigates away and back, the component still needs to reconnect correctly.

Fix the pair instead:

```
useEffect(() => {
  const connection = createConnection(roomId)
  connection.connect()

  return () => connection.disconnect()
}, [roomId])
```

Operations such as purchases and form submissions belong in event handlers, where a specific action caused them. Server endpoints for important operations should also handle retries and duplicates safely.

Strict Mode behavior is development-only, but the bugs it reveals are real.

Log connect and disconnect calls while changing rooms and mounting the component. Every started connection should have a matching cleanup.

Extract a small custom Hook

Move repeated stateful behavior into a named function while keeping the rendered markup in components.

A custom Hook packages reusable stateful behavior behind a clear name.

```
function useOnlineStatus() {
  const [online, setOnline] = useState(navigator.onLine)

  useEffect(() => {
    function handleOnline() {
      setOnline(true)
    }

    function handleOffline() {
      setOnline(false)
    }

    window.addEventListener('online', handleOnline)
    window.addEventListener('offline', handleOffline)

    return () => {
      window.removeEventListener('online', handleOnline)
      window.removeEventListener('offline', handleOffline)
    }
  }, [])

  return online
}
```

A component can now use the behavior without knowing the subscription details:

```
const online = useOnlineStatus()
```

Extract a custom Hook when several components need the same synchronization or state transition logic, or when a named abstraction makes one complex component easier to read.

Keep the name specific. `useOnlineStatus` explains a result. `useStuff` hides it.

A custom Hook shares logic, not one state value. Two calls have independent Hook state, although both may subscribe to the same browser source.

Do not extract every pair of `useState` calls. A custom Hook should create a meaningful boundary, not move code to another file without improving the model.

Use the Hook from two components and toggle offline mode in DevTools. Both should update, and removing one component should clean up only its listeners.

Check your understanding: Hooks and effects

Check useState, functional updates, Hook call order, effects, dependencies, cleanup, and synchronization.

Answer each question before continuing. Read the explanation after every answer, including the

ones you get right.

Take this interactive quiz online

Events and forms

Respond to interactions and manage accessible form inputs.

Handle an event

Pass a function to a JSX event prop and let React call it when the interaction occurs.

Event handlers run in response to a user interaction.

```
function SaveButton() {
  function handleClick() {
    console.log('Saved')
  }

  return <button onClick={handleClick}>Save</button>
}
```

Event prop names use camelCase. Pass the function as `onClick={handleClick}`.

This is wrong:

```
<button onClick={handleClick()}>Save</button>
```

It calls the function during rendering and passes its return value to `onClick`.

Put work caused by the interaction in the handler: update state, submit data, or call a parent callback. Keep rendering pure.

Use the semantic element for the interaction. A `button` already supports keyboard activation, focus, and disabled behavior. A clickable `div` makes you rebuild those features.

Name handlers after the event or intent: `handleClick`, `handleSave`, or `onSave`. A clear name makes the data flow easier to follow.

Add a console log inside the component body and another inside the handler. Reload and click once. Notice which log belongs to rendering and which belongs to the event.

Pass data to an event handler

Wrap a call in a small arrow function when the handler needs an item id or another render-time value.

Wrap a call in an arrow function when the handler needs data from the current render:

```
function Task({ task, onRemove }) {
  return (
    <button onClick={() => onRemove(task.id)}>
      Remove {task.title}
    </button>
  )
}
```

The arrow function is the event handler. React calls it later. It then calls `onRemove` with the task ID captured by this render.

Do not write this:

```
<button onClick={onRemove(task.id)}>Remove</button>
```

That calls `onRemove` while rendering.

Pass a stable identifier rather than the list index. If items move, an index can point to a different task by the time the parent updates its state.

You can pass the browser event too when needed:

```
onClick={event => onRemove(task.id, event)}
```

Most application handlers need the domain value more than the raw event. Keep the child callback focused on the intent.

Reorder a task list, remove one item, and confirm the correct ID reaches the parent.

Understand event propagation

Recognize that nested event handlers can both run and stop propagation only when the interaction should not reach an ancestor.

Browser events normally bubble from the target through its ancestors.

```
function TaskRow() {
  return (
    <article onClick={() => console.log('Open task')}>
      <h2>Update the docs</h2>
      <button onClick={() => console.log('Delete task')}>
        Delete
      </button>
    </article>
  )
}
```

Clicking Delete runs the button handler, then the article handler. That may open a task while deleting it.

If the interactions are intentionally separate, stop propagation in the nested handler:

```
function handleDelete(event) {
  event.stopPropagation()
  deleteTask()
}
```

Use this deliberately. Global propagation blocking can break analytics, keyboard behavior, and parent-level event handling.

Often the better design is not to make the entire container clickable. Use a real link for navigation and a separate button for deletion. Clear semantics can remove the propagation conflict.

`preventDefault()` solves another problem. It prevents the browser's default action, such as form navigation or following a link. It does not stop the event from reaching ancestors.

Log `event.target` and `event.currentTarget` in both handlers. The target is where the event began. The current target is the element whose handler is running.

Control an input with state

Make React state the current source of truth by pairing the input value with an onChange handler.

A controlled input gets its current value from React state.

```
function TaskForm() {
  const [title, setTitle] = useState('')

  return (
    <label>
      Task title
      <input
        name="title"
        value={title}
        onChange={event => setTitle(event.target.value)}
      />
    </label>
  )
}
```

Each edit follows the React cycle:

1. The browser reports the edit and React calls the `onChange` handler.
2. The handler requests a state update.
3. React renders with the new `title`.
4. The input receives that value.

State is the source of truth. This makes it easy to display a character count, format a value, or disable submission according to the same data.

Do not pass `value` without `onChange` unless the input is intentionally read-only. React will keep restoring the prop value, so typing appears broken.

Initialize text inputs with a string, usually `''`. Switching between `undefined` and a string changes between uncontrolled and controlled behavior and produces warnings.

Controlling every keystroke causes the owner component to render. Keep state close to the form so unrelated expensive sections do not rerender. Optimize only after measuring a real problem.

Add a live character count and confirm the input, count, and state always agree.

Submit a form

Handle the form submit event so keyboard submission and a submit button use one accessible path.

Handle submission on the form, not only on the button. A form can be submitted several ways: clicking the button, pressing Enter in a field, or using assistive technology. The form's `submit` event is the one place all of those paths meet, while a `click` handler on the button catches only one of them.

```
function TaskForm({ onSave }) {
```

```
  const [title, setTitle] = useState('')

  function handleSubmit(event) {
    event.preventDefault()

    const trimmedTitle = title.trim()
    if (!trimmedTitle) return

    onSave(trimmedTitle)
    setTitle('')
  }

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="title">Task title</label>
      <input
        id="title"
        name="title"
        value={title}
        onChange={event => setTitle(event.target.value)}
        required
      />
      <button type="submit">Add task</button>
    </form>
  )
}
```

The form's submit event covers pointer clicks and keyboard submission. The button remains a real submit button, so it keeps its native behavior and semantics.

Call `preventDefault()` because this component handles the result without native navigation. Without it, the browser performs a full page load on submit, and your React state disappears with the page. If the form seems to ”flash” and reset, a missing `preventDefault()` is the first thing to check. For a progressively enhanced server form, you may keep the normal `action` and `method` instead and let the browser navigate.

The handler trims the value and returns early when it is empty. React does not validate anything for you; the handler owns that logic. Browser validation such as `required` improves feedback before submission ever fires. It is not a security boundary. A server endpoint must validate and authorize every request again.

When the save is asynchronous, do not clear the form before it succeeds. Clearing early destroys the user's input if the request fails. Keep the current value, show a pending state, and display a useful error if the request fails.

Submit once by clicking and once by pressing Enter. Both paths should call the same handler, and you can prove it with a log line in `handleSubmit`.

Know when an input can stay uncontrolled

Let the DOM hold an initial value when React does not need to render every edit.

An uncontrolled input keeps its current value in the DOM instead of React state.

```
function SearchForm() {
  function handleSubmit(event) {
    event.preventDefault()

    const data = new FormData(event.currentTarget)
    console.log(data.get('query'))
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Search
        <input name="query" defaultValue="React" />
      </label>
      <button type="submit">Search</button>
    </form>
  )
}
```

`defaultValue` supplies the initial value. Editing after that belongs to the DOM. Changing the `defaultValue` prop later does not replace the current field value.

Use an uncontrolled input when the value is needed only on submission and the rest of the interface does not react to every edit. Native forms and `FormData` work well here.

Use controlled state when another part of the interface needs the current value, such as a live preview, character count, or dependent field.

File inputs are uncontrolled because browser security rules prevent application code from setting a local file path.

Do not mix `value` and `defaultValue`. Choose which layer owns the current value.

Change the default prop after typing and observe that the DOM keeps the edit. Then rebuild the same field as controlled state and compare the behavior.

Check your understanding: events and forms

Check event props, handler references, propagation, default actions, controlled fields, and form submission.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Rendering and data flow

Render conditions and lists, lift state, and debug the result.

Render a condition

Use normal JavaScript to choose which JSX a component returns.

Use ordinary JavaScript conditions to choose JSX.

Return early when the whole screen state changes:

```
function Account({ user }) {
  if (!user) {
    return <LoginForm />
  }

  return <Dashboard user={user} />
}
```

Use a ternary for two small alternatives:

```
<p>{saved ? 'Saved' : 'Unsaved changes'}</p>
```

Use `&&` when the false branch should render nothing:

```
{error && <p role="alert">{error}</p>}
```

Be careful with numbers. `items.length && <List />` renders `0` for an empty list. Write `items.length > 0 && <List />` when that is the real condition.

Do not hide inaccessible markup with CSS when it should not exist in the interaction tree. Conditional rendering can remove it from both the DOM and accessibility tree.

Avoid deeply nested ternaries. Calculate a named variable or extract a component when the states become hard to scan.

Render loading, error, empty, and success states for one list. Make each state explicit and reachable in a test.

Render a list with map

Turn an array of data into an array of elements or components.

Transform data into elements with `map()`:

```
function TaskList({ tasks }) {
  return (
    <ul>
      {tasks.map(task => (
        <li key={task.id}>{task.title}</li>
      ))}
    </ul>
  )
}
```

React renders each returned element as a sibling. The key helps React match each task with its previous element on the next render.

Filter data before mapping when the interface shows a subset:

```
const openTasks = tasks.filter(task => !task.done)
```

Keep the transformation pure. Do not delete tasks, update state, or sort the original state array while rendering. Copy before sorting:

```
const sortedTasks = [...tasks].sort(compareTasks)
```

Render a useful empty state when the array contains nothing. An empty `ul` often gives no explanation.

When a list item grows, extract a `Task` component and pass the item plus intent callbacks. Keep the key on the element returned directly from `map()`.

Add, remove, and reorder tasks. The displayed order should come entirely from the array used by this render.

Give list items stable keys

Help React match each rendered item with the same data item across insertions, removals, and reordering.

A key identifies an item among its siblings across renders.

```
{tasks.map(task => (
  <Task key={task.id} task={task} />
))}
```

When tasks move, React uses each ID to keep component and DOM state with the correct task.

Array indexes are unsafe when items can be inserted, removed, sorted, or filtered. If the first task is removed, the item now at index zero can inherit the previous first component's input state.

Generate IDs when data is created, not while rendering. `key={Math.random()}` changes identity on every render and forces React to recreate the subtree.

Keys need to be unique only among siblings. The same task ID can appear in a separate list.

`key` is for React and is not passed as a normal prop. Pass `id={task.id}` separately when the component needs it.

Build a list with editable inputs, then sort it. Compare stable IDs with index keys and watch which text stays with each item.

Group elements with a fragment

Return siblings without adding an unnecessary wrapper element to the DOM.

A React component can only return one single element. When you want to render two siblings, the quick fix is wrapping them in a `div`. Do that in every component and the page fills with wrapper elements that exist only to satisfy React.

The extra `div` is not always harmless. Some HTML structures forbid stray elements between their children: a definition list expects `dt` and `dd`, a table row expects cells. A wrapper in the wrong place produces invalid HTML, and it can also interfere with CSS layouts built on direct-child selectors, flexbox, or grid.

A **Fragment** groups sibling elements without adding a DOM wrapper:

```
function BlogPostExcerpt({ title, description }) {
  return (
```

```
    <>
      <h1>{title}</h1>
      <p>{description}</p>
    </>
  )
}
```

The empty tags `<>...</>` are the shorthand syntax for a fragment. React receives one return value, while the browser gets `h1` and `p` as direct siblings with nothing around them. Fragments do not appear in the DOM at all.

There is also an explicit form, imported from React. Older codebases write `React.Fragment`, which is the same component. Use the shorthand by default.

The explicit form earns its keep in one common case: when the group needs a `key`. This happens when a list renders pairs of elements, like the name and role rows of a definition list:

```
import { Fragment } from 'react'

function PeopleList({ people }) {
  return (
    <dl>
      {people.map(person => (
        <Fragment key={person.id}>
          <dt>{person.name}</dt>
          <dd>{person.role}</dd>
        </Fragment>
      ))}
    </dl>
  )
}
```

The short `<>...</>` syntax cannot receive a key or any other attribute, so this is where you spell out `Fragment`.

One caution before you replace every wrapper. Do not use a fragment when a semantic wrapper would improve the document. A group that forms a section may need `section`; navigation may need `nav`. The fragment is for groups that need no element of their own, not a way to avoid thinking about HTML structure.

Inspect the DOM with and without a `div` wrapper. Choose the version that preserves the intended HTML structure.

Lift shared state up

Move one piece of state to the nearest common owner when two components must stay in sync.

When two components need the same changing value, move that state to their closest common parent.

```
function TemperatureCalculator() {
  const [celsius, setCelsius] = useState('')

  return (
```

```
    <>
      <TemperatureInput
        label="Celsius"
        value={celsius}
        onChange={setCelsius}
      />
      <p>Fahrenheit: {convertToFahrenheit(celsius)}</p>
    </>
  )
}
```

The parent passes the current value down. The child reports edits through `onChange`. The converted value is calculated during rendering rather than stored as second state.

This creates one source of truth. Every child receives a snapshot derived from the same owner.

Do not lift every local detail. Whether a dropdown is open may matter only inside that dropdown. Moving all state to the page creates unnecessary prop wiring and broader renders.

Lift state when siblings need coordination or a parent must make decisions from the value. Keep it local otherwise.

Build two inputs that edit the same text. First give each local state, then lift it and confirm both always agree.

Follow one-way data flow

Trace state from its owner down through props and trace interactions back up through callback props.

Unidirectional data flow means data has one, and only one, way to move through the application. In React, that direction is down: state is passed from its owner to child components through props, and nothing flows back up on its own.

What travels upward instead are events. A child receives a callback prop and calls it to report what happened:

```
function TaskPage() {
  const [tasks, setTasks] = useState(initialTasks)

  function handleToggle(id) {
    setTasks(tasks => tasks.map(task =>
      task.id === id
        ? { ...task, done: !task.done }
        : task
    ))
  }

  return <TaskList tasks={tasks} onToggle={handleToggle} />
}
```

Inside `TaskList`, a child renders each task and reports clicks through the callback prop:

```
<button onClick={() => onToggle(task.id)}>
```

```
  {task.title}
</button>
```

Follow one interaction through this component:

1. `TaskPage` owns the task state.
2. It passes task snapshots down as props.
3. A child reports intent with `onToggle(id)`.
4. The owner updates state.
5. React renders new props down the tree.

The child does not mutate the task it received or search the DOM for another component. It only reports what happened. The owner remains the one source of truth.

Why organize an application this way? Because it limits where data can come from. When every value has exactly one owner and one downward path, you have more control over your data, and debugging becomes tracing: you know what is coming from where. Two-way bindings, where a child edits shared data directly, remove that guarantee.

The rule has a structural consequence worth memorizing. Changing state on a component affects only that component and its children. It never affects the parent, the siblings, or anything else in the tree. This is why state often gets moved up: when two components need the same value, the state must live in a common ancestor so it can flow down to both.

This model also gives you a debugging procedure. When a value is surprising, find the component that owns it. Follow the prop downward to where it renders and the callback upward to where it changes. React DevTools shows both points, so the search is short.

Draw the owner, prop, child, callback, and next render for one interaction before changing code.

Render an overlay with a portal

Place a component child in another DOM container while keeping it in the same React component tree.

A portal renders children into another DOM node while keeping them in the same React tree.

```
import { createPortal } from 'react-dom'

function Modal({ children }) {
  const modalRoot = document.getElementById('modal-root')
  return createPortal(children, modalRoot)
}
```

This is useful when an overlay must escape a container's clipping or stacking context. The modal DOM can live near the end of `body` while its component remains a child of the page in React.

Context still follows the React tree. React events also propagate through React parents, not only DOM parents.

A portal only changes placement. It does not create accessible dialog behavior.

A modal still needs an accessible name, focus moved inside, Escape and close-button handling, background interaction rules, and focus returned to the opener.

Use the native `dialog` element where it fits; it provides useful browser behavior that a generic portal

does not.

Inspect the component tree and DOM tree for a portal. Confirm the same content has different parents in the two views.

Debug component data flow

Use component inspection, logs, and small reproductions to locate the owner of an incorrect value or repeated update.

Start from the visible symptom and trace data to its owner.

Use a fixed order:

1. Inspect the final DOM and accessibility state.
2. Find the component that rendered it in React DevTools.
3. Check its props, state, and Hooks.
4. Find the owner of the first wrong value.
5. Inspect the event or Effect that updates it.

Add focused logs with context:

```
console.log('TaskList render', {
  filter,
  taskCount: tasks.length
})
```

A log inside the component body records rendering. A log inside an event handler records an interaction. Strict Mode may repeat development renders, so one log does not always mean one user action.

If data is correct but the screen is wrong, switch to the Elements, Accessibility, and CSS panels. React cannot fix invalid layout or a missing label by changing state.

Do not add state to compensate for a wrong derived value. Fix the first incorrect value or ownership decision.

Reduce the bug to the smallest component tree that still shows it. Then write down the owner and each transformation before editing.

Build a React task list

Combine the complete course in a small application with reusable components, state ownership, forms, filtering, and persistence.

Build a task list that brings the complete React model together.

The application needs:

- an add form
- completion checkboxes
- remove buttons
- all, open, and completed filters
- browser persistence

Choose the owner

Keep the task array and current filter in `App`. The form, filter controls, and list all need that data or need to request changes to it.

Pass snapshots down:

```
<TaskForm onAdd={handleAdd} />
<TaskFilters value={filter} onChange={setFilter} />
<TaskList
  tasks={visibleTasks}
  onToggle={handleToggle}
  onRemove={handleRemove}
/>
```

Calculate `visibleTasks` during rendering. Do not store a filtered copy in state.

Update the array immutably with updater functions. Generate a stable ID when adding a task. Use that ID for the React key and for toggle and remove events.

Build the form from HTML

Use a labeled input and a real submit button. Handle `onSubmit` on the form so Enter works. Reject a title containing only whitespace.

Keep the input value after an asynchronous failure. Clear it only when adding succeeds.

Add persistence deliberately

Local storage is an external browser system. Read the initial value once, parse it defensively, and fall back to an empty array when stored data is invalid.

Use one Effect to synchronize the current task array back to storage:

```
useEffect(() => {
  localStorage.setItem('tasks', JSON.stringify(tasks))
}, [tasks])
```

Do not put the filtered list in storage. It can always be derived from tasks and the current filter.

Verify the model

Test keyboard submission, an empty list, duplicate titles, every filter, item removal, reordered items, a page reload, and invalid JSON in storage.

Use React DevTools to identify the state owner. Use the Profiler only if an interaction feels slow. The finished application should be explainable as props down, events up, state update, and next render.

Check your understanding: rendering and data flow

Check conditions, lists, keys, fragments, lifted state, one-way data flow, state identity, and portals.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/react/.