

2 Advanced component patterns

This chapter covers

- Providing global state with the Provider pattern
- Managing complex component structures with the Composite pattern
- Creating clean components with the Summary pattern

The construction world, with projects ranging from sky-touching skyscrapers to peaceful neighborhood homes, adheres to a universal set of principles. In both architecture and construction, regardless of project size, there is a steadfast commitment to core engineering principles, which include designing load-bearing structures, selecting appropriate materials, and ensuring overall stability and safety.

In the digital realm, React development is committed to its own set of construction principles. Although the outer aesthetics and specific materials may differ, the underlying architectural principles remain constant.

Within React, as in software design in general, these principles are embodied in design patterns. This chapter will take an in-depth look at three foundational patterns used in modern React applications:

- Provider
- Composite
- Summary

Much like the blueprints and load-bearing structures in physical construction, these patterns, shown in figure 2.1, provide the framework for building stable and scalable React projects.

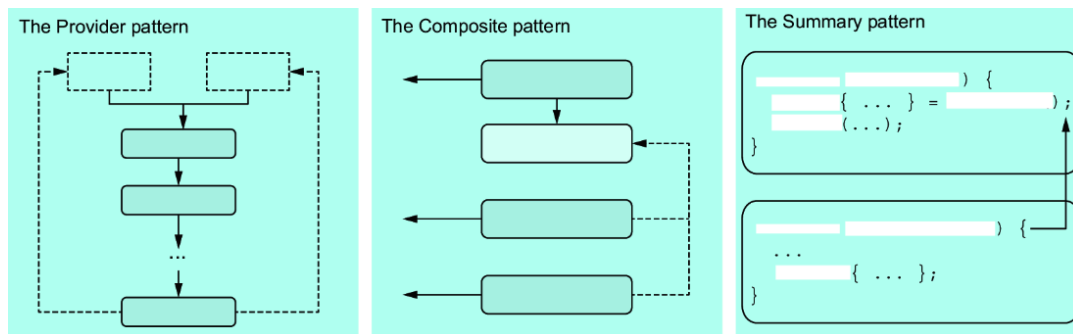


Figure 2.1 The three design patterns that we'll take a closer look at in this chapter, illustrated as though they're construction blueprints. I'll go over each illustration in detail in the appropriate section.

Consider figure 2.1 to be a metaphorical construction diagram that introduces these crucial patterns. Throughout this chapter, we will explore how these patterns enable you to construct resilient React applications that are capable of withstanding various challenges.

Just as architects and builders across the globe rely on consistent physical and engineering principles, React developers use patterns to create a diverse array of interactive experiences. By adhering to these foundational strategies, you can build React applications that are as varied and innovative as the world's architectural wonders. So gear up with your hard hat and tools and embark on a journey through the intricate construction site of React's advanced component patterns, where solidity and stability guide us to create robust digital structures.

Note The source code for the examples in this chapter is available at <https://reactlikea.pro/ch02>.

2.1 The Provider pattern

In this section, we delve into the Provider pattern, building on your existing knowledge of React context. We'll explore how to use the Provider pattern to manage multiple related values, such as state values and their corresponding setters. This approach (figure 2.2) represents a significant advancement beyond basic use of React context, offering enhanced flexibility and efficiency in state management.

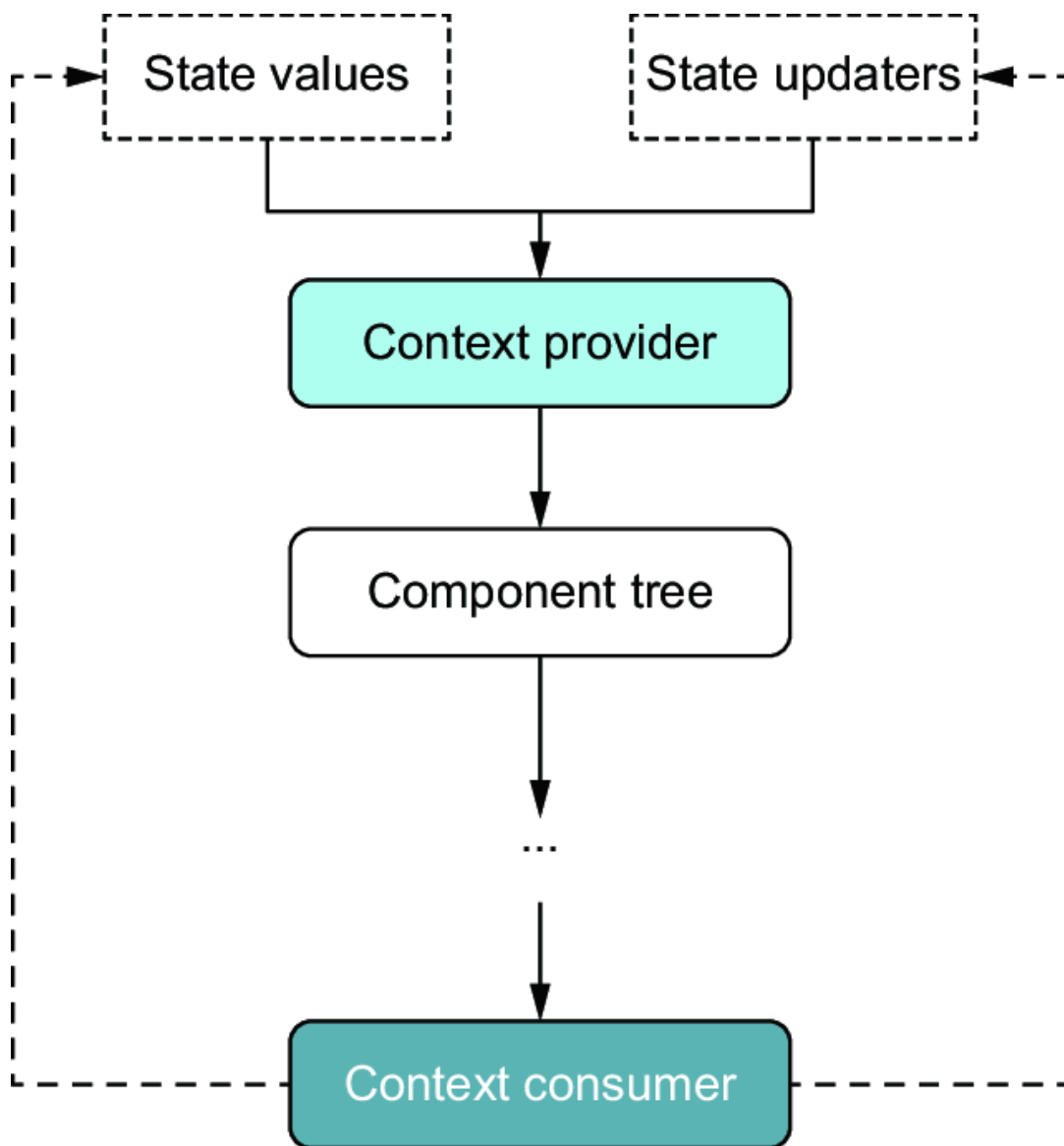


Figure 2.2 This illustration highlights the initial step of using the Provider pattern to handle multiple related values. It shows a basic context setup wherein the provider (light gray background) encapsulates both state values and updaters (dashed boxes), providing a comprehensive view of how we extend the conventional use of React context. The context consumer (dark gray background) can be at any depth inside the component tree below the consumer but can still easily access the values and updaters provided by the context (dashed arrows).

Our exploration is structured to facilitate hands-on discovery through the following stages:

- *Context with multiple values*—Starting with a straightforward example, we'll see how to wrap a child component in a provider that handles multiple related state values and setters. This foundational step showcases the basic extension of React context beyond singular data points.

- *Dedicated component for context management* —Next, we evolve our approach by creating a dedicated component for the provider. This refinement addresses the intricacies of managing multiple state aspects and illustrates a more structured and maintainable way to handle complex contexts.
- *Selectability for performance optimization* —Finally, we introduce selectability to our Provider pattern. This advanced technique focuses on minimizing re-renders, especially for components with stable content. It demonstrates how selective data flow can significantly enhance the performance of your React applications.

By the end of this section, you will not only have deepened your understanding of React context but also gained practical skills by implementing the Provider pattern for complex state management. This journey from basic implementation to sophisticated techniques will empower you to optimize your React applications, ensuring that they are both performant and maintainable.

2.1.1 Inventing a provider

A common approach is to use a context as a delivery mechanism for stateful values *and* setters. Suppose that we have a website with dark mode and light mode and a button in the header that can toggle between the two modes. All relevant components look at the current state and change their design depending on this state value.

We want to put two things in the state: a value that tells us whether we are in dark mode (`isDarkMode`) and a function that allows a button to toggle between the two modes (`toggleDarkMode`). We can put these two values in a single object and stuff that object into the context as the value. Figure 2.3 shows this system, which we'll implement in listing 2.1.

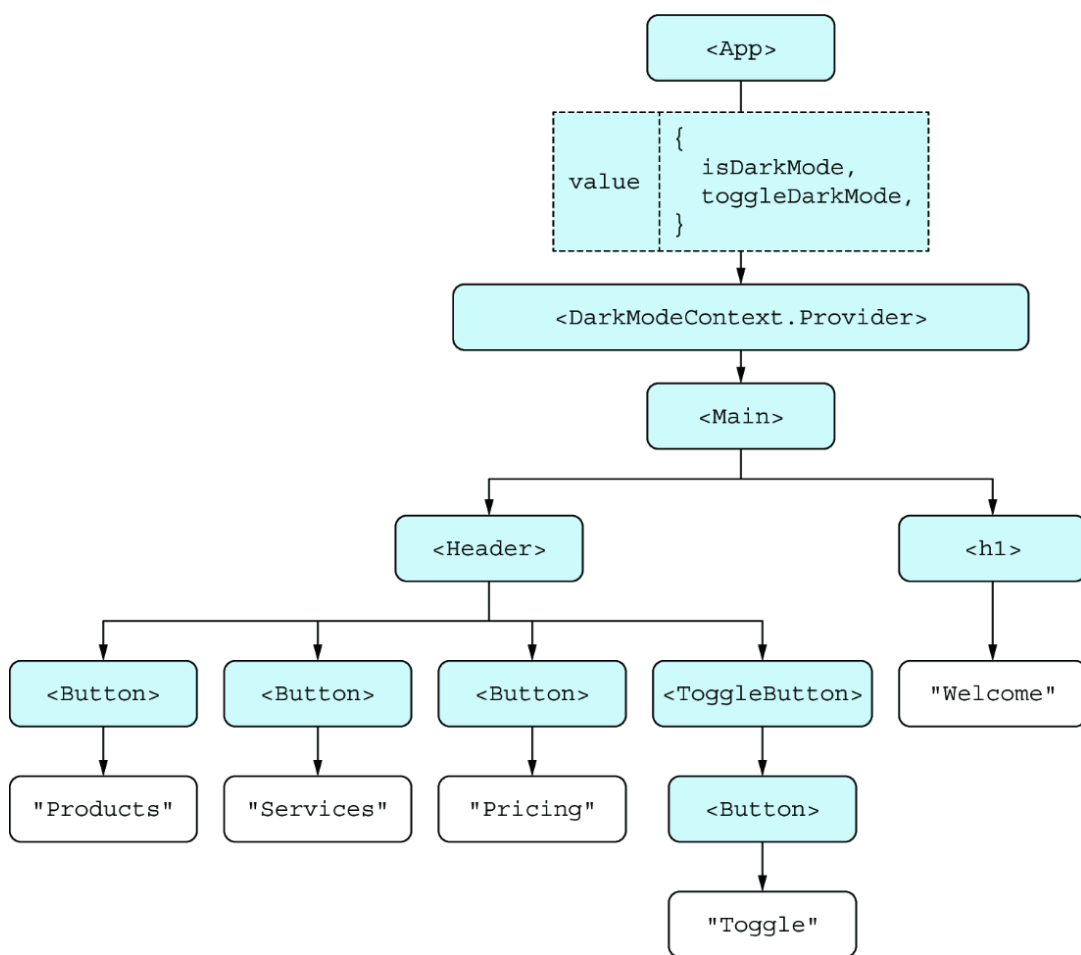


Figure 2.3 A document tree sketch of our website with a dark mode/light mode toggle. Note how we pass in an object to the context provider (dashed box), which we can deconstruct and use wherever we need either of the two values in the components below the provider in the document tree.

Listing 2.1 Dark mode with context

```
import { useContext, useState, createContext, memo } from "react";
const DarkModeContext = createContext({});    #1
function Button({ children, ...rest }) {
  const { isDarkMode } = #2
    useContext(DarkModeContext);    #2
  const style = {
    backgroundColor: isDarkMode ? "#333" : "#CCC",
    border: "1px solid",
    color: "inherit",
  };
  return (
    <button style={style} {...rest}>
      {children}
    </button>
  );
}
function ToggleButton() {
  const { toggleDarkMode } = useContext(DarkModeContext);    #3
  return <Button onClick={toggleDarkMode}>Toggle mode</Button>;
}
const Header = memo(function Header() {
  const style = {
    padding: "10px 5px",
    borderBottom: "1px solid",
    marginBottom: "10px",
    display: "flex",
    gap: "5px",
    justifyContent: "flex-end",
  };
  return (
    <header style={style}>
      <Button>Products</Button>
      <Button>Services</Button>
      <Button>Pricing</Button>
      <ToggleButton />
    </header>
  );
});
const Main = memo(function Main() {    #4
  const { isDarkMode } = #5
```

```

↪ useContext(DarkModeContext); #5
const style = {
  color: isDarkMode ? "white" : "black",
  backgroundColor: isDarkMode ? "black" : "white",
  margin: "-8px",
  minHeight: "100vh",
  boxSizing: "border-box",
};
return (
  <main style={style}>
    <Header />
    <h1>Welcome to our business site!</h1>
  </main>
);
});
export default function App() {
  const [isDarkMode, setDarkMode] = #6
    useState(false); #6
  const toggleDarkMode = #6
    () => setDarkMode((v) => !v); #6
  const contextValue = { #7
    isDarkMode, #7
    toggleDarkMode #7
  };
  return (
    <DarkModeContext.Provider
      value={contextValue} #8
    >
      <Main />
    </DarkModeContext.Provider>
  );
}

```

#1 This time, we initialize our context with an empty object. We always have a context at the root of the application, so the default values will never be used.

#2 In these two locations, we use only the isDarkMode flag from the context.

#3 In the toggle button, we use only the toggleDarkMode function from the context.

#4 We memoize the main component.

#5 In these two locations, we use only the isDarkMode flag from the context.

#6 In the main application component, we define the two values that go into our context.

#7 We put these two values together in a single object.

#8 We use this single object as the value for our context provider.

EXAMPLE: DARK-MODE

This example is in the `ch02/dark-mode` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch02/dark-mode
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: <https://reactlikea.pro/ch02-dark-mode>. You can observe this website in figure 2.4.

Note In React 19, context providers are created in JSX by typing `<MyContext value={...}>` rather than `<MyContext.Provider value={...}>`. Furthermore, contexts can be consumed by using the `use()` function rather than the `useContext()` hook. The new `use()` function is not a regular hook and does not have to obey hook rules, so it can be used conditionally, but it doesn't give you any additional capability—only a different API. Creating and using contexts are otherwise the same in React 19 except for these slight syntax simplifications. I will be using the old syntax throughout this book.

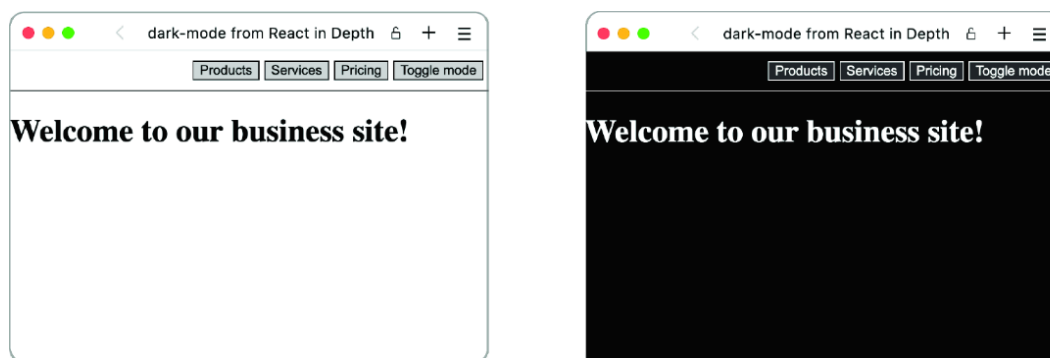


Figure 2.4 Our website in both light mode and dark mode. Both examples look pretty decent and even somewhat chic!

The important things to note here are how we give this context two different properties in the definition of the `<App />` component in listing 2.1 and how we memoize the first components inside the context provider in the definition of `<Main />`. This memoization is very

important because our main `App` component will re-render every time the context changes, which is every time the dark mode flag toggles (because the state updates). We don't want all the other components to re-render just because the context does, however. In this instance, the `Main` component consumes the context, so it will re-render every time the context updates, but the `Header` does not consume the context, so it should not re-render. With our use of memoization, it doesn't, which is perfect.

We don't have to stop there. We can put a whole bunch of properties and functions in the context value.

2.1.2 Creating a dedicated provider component

The previous version of the dark mode application in listing 2.1 is fully functional, but we can do a bit better. The main application component is a bit crowded with the state value, toggle function, and context provider, so let's clean it up. Instead of this code,

```
function App() {
  const [isDarkMode, setDarkMode] = useState(false);
  const toggleDarkMode = () => setDarkMode(v => !v);
  const contextValue = { isDarkMode, toggleDarkMode };
  return (
    <DarkModeContext.Provider value={contextValue}>
      <Main />
    </DarkModeContext.Provider>
  );
}
```

suppose that we have this code:

```
function App() {
  return (
    <DarkModeProvider>
      <Main />
    </DarkModeProvider>
  );
}
```

First, the second component is much more elegant. We remove the logic about what goes inside the actual context from the main application, but we also get one additional benefit: this new `<App />` component is not stateful, so it never re-renders. Because it never re-renders, it never causes `<Main />` to re-render.

Before, `<App />` was stateful and caused re-renders of the `Main` component, so we had to wrap that component in `memo()` to avoid unnecessary re-renders, but we don't have to anymore. One additional optimization we can make is to simplify these calls:

```
const ... = useContext(DarkModeContext);
```

We can create a custom hook that returns the context contents, so this line becomes

```
const ... = useDarkMode();
```

With both changes, we get the result in the following listing.

Listing 2.2 Dark mode with a dedicated provider

```
import { useContext, useState, createContext, memo } from "react";
const DarkModeContext = createContext({});
function Button({ children, ...rest }) {
  const { isDarkMode } = useDarkMode();    #1
  ...
}
function ToggleButton() {
  const { toggleDarkMode } = useDarkMode();    #1

  return <Button onClick={toggleDarkMode}>Toggle mode</Button>;
}
const Header = memo(function Header() {
  ...
});
function Main() {    #2
  const { isDarkMode } = useDarkMode();    #1
  ...
}
function DarkModeProvider({ children }) {    #3
  const [isDarkMode, setDarkMode] = useState(false);
  const toggleDarkMode = () => setDarkMode((v) => !v);
  const contextValue = { isDarkMode, toggleDarkMode };
  return (
    <DarkModeContext.Provider value={contextValue}>
      {children}
    </DarkModeContext.Provider>
  );
}
function useDarkMode() {    #4
  return useContext(DarkModeContext);
}
export default function App() {
  return (
    <DarkModeProvider>    #5
      <Main />    #5
    </DarkModeProvider>    #5
  );
}
```

- #1 Uses the custom hook to access the context contents
- #2 Defines the main component without memoization
- #3 Creates a new dedicated provider component that wraps its children in the context provider
- #4 Creates a new custom hook to access the provided context
- #5 Returns a much more elegant JSX in the root app component

EXAMPLE: DARK-MODE-DEDICATED

This example is in the `ch02/dark-mode-dedicated` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch02/dark-mode-dedicated
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: <https://reactlikea.pro/ch02-dark-mode-dedicated>.

2.1.3 Avoiding rendering everything

The context provider in the preceding example has a minor suboptimal problem: *all* components consuming a specific context will re-render when *any* value inside that context changes. This situation occurs because now our context is a complex object with multiple properties, but React doesn't care; it sees only that the context value changes, so every component using that context will be re-rendered.

Our toggle component never needs to re-render, however, because it uses a function that can be memoized to be completely stable. The reason is that the `toggleDarkMode` function does not depend on the current value of the context. Unfortunately, we cannot tell React to re-render only a specific component when specific properties of a context update. At least, we cannot do that *yet*. That capability was expected to come with React 19 beta but didn't make it; it's hopefully coming in a future update.

If we want to avoid re-rendering every context consumer unnecessarily, we need to use an external library. One such library, called `use-context-selector`, allows us to not use an entire context every time.

Instead, we can specify the specific attribute of the context that we are interested in (we select the relevant property—hence, the `selector` part of the name). Then React will re-render our component only when that specific property changes.

To use the `use-context-selector` package correctly, we also need to create our context with this package. We cannot use the regular context as created by `createContext` in the React package; we have to use the `createContext` function provided by the `use-context-selector` package. The custom hook for accessing the context takes a `selector` function, like so:

```
function useDarkMode(selector) {  
  return useContextSelector(DarkModeContext, selector);  
}
```

We pass this new argument, `selector`, straight to the `useContextSelector` hook. This custom hook still makes sense, as it removes the need to reference the context every time. Let's implement this updated and more optimized version of our dark mode–toggling website in the following listing.

Listing 2.3 Dark mode with context selector

```
import { useState, useCallback, memo } from "react";
import {
  #1
  createContext,
  #1
  useContextSelector,
  #1
} from "use-context-selector";
const DarkModeContext = createContext({});
function Button({ children, ...rest }) {
  const isDarkMode =
    #2
    useDarkMode((ctx) => ctx.isDarkMode);
  #2
  ...
}
function ToggleButton() {
  const toggle = useDarkMode((ctx) => ctx.toggle);
  #3
  return <Button onClick={toggle}>Toggle mode</Button>;
}
const Header = memo(function Header() {
  ...
});
function Main() {
  const isDarkMode =
    #3
    useDarkMode((ctx) => ctx.isDarkMode);
  #3
  ...
}
function DarkModeProvider({ children }) {
  const [isDarkMode, setDarkMode] = useState(false);
  const toggle =
    #4
    useCallback(() => setDarkMode((v) => !v), []);
  #4
  const contextValue = { isDarkMode, toggle };
  #5
  return (
    <DarkModeContext.Provider value={contextValue}>
      {children}
    </DarkModeContext.Provider>
  );
}
function useDarkMode(selector) {
  #6
  return useContextSelector(
    #6
    DarkModeContext,
    #6
    selector
    #6
  );
  #6
}
```

```
export default function App() {  
  return (  
    <DarkModeProvider>  
      <Main />  
    </DarkModeProvider>  
  );  
}
```

#1 We have changed only a few things this time around. First, we imported two functions from the use-context-selector package.

#2 Whenever we need a value from the context, we use the new useDarkMode hook, which now takes a selector.

#3 Whenever we need a value from the context, we use the new useDarkMode hook, which now takes a selector.

#4 To avoid re-renders, we memoize our toggle function by using useCallback.

#5 We create and initialize the context value the same as before from the two parts.

#6 In the useDarkMode hook, we need to pass the selector argument to that new useContextSelector hook from the third-party package.

EXAMPLE: DARK-MODE-SELECTOR

This example is in the `ch02/dark-mode-selector` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch02/dark-mode-selector
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: <https://reactlikea.pro/ch02-dark-mode-selector>.

The result is the same website we had before with the same functionality, but now the `ToggleButton` never re-renders because it uses only a stable value from the context. Because the context never updates, there's no need to re-render the component. The two components listening for the `isDarkMode` flag inside the context will still re-render every time the flag updates because we select that exact property in the `useDarkMode` hooks in those two components.

This approach might seem like an overoptimization at this point because we're talking about whether a single component updates a few extra times or not. In a large application with many contexts, however, these extra updates add up quickly! So if you are using contexts to share common functionality throughout your application, you should be using `useContextSelector` rather than the normal `useContext` hook—unless React implements the selection logic as part of the normal `useContext` hook by the time you read this book.

2.1.4 Creating beautifully typed selectable contexts with the recontextual tool

In the previous examples, we used only JavaScript, not TypeScript. I'll talk a lot more about TypeScript in chapters 5 and 6 and in later chapters. For now, I'll say only that some of these patterns are quite tricky to type in an elegant way.

To make selectable contexts easier to type, I created a small package, `recontextual`, that wraps `use-context-selector` and provides a simple way to create well-typed selectable contexts with minimal typing effort. Without getting into too much detail, let's see how the dark mode application looks in TypeScript, using the new library in the following listing.

Listing 2.4 Dark mode with a typed selector

```
import {
  useState,
  useCallback,
  memo,
  PropsWithChildren,
  ComponentPropsWithoutRef,
} from "react";
import recontextual from "recontextual";    #1
interface DarkModeContext {                #2
  isDarkMode: boolean;                     #2
  toggle: () => void;                       #2
}                                           #2
const [Provider, useDarkMode] =           #3
  recontextual<DarkModeContext>();          #3
function Button({
  children,
  ...rest
}: PropsWithChildren<ComponentPropsWithoutRef<"button">>) {
  const isDarkMode =
    useDarkMode((ctx) => ctx.isDarkMode);    #4
  ...
}
function ToggleButton() {
  const toggle = useDarkMode((ctx) => ctx.toggle); #4
  return <Button onClick={toggle}>Toggle mode</Button>;
}
const Header = memo(function Header() {
  ...
});
function Main() {
  const isDarkMode =
    useDarkMode((ctx) => ctx.isDarkMode);    #5
  ...
}
function DarkModeProvider({ children }: PropsWithChildren) {
  const [isDarkMode, setDarkMode] = useState(false);
  const toggle = useCallback(() => setDarkMode((v) => !v), []);
  const contextValue = { isDarkMode, toggle };
  return <Provider value={contextValue}>{children} </Provider>;    #6
}
```

```
export default function App() {  
  return (  
    <DarkModeProvider>  
      <Main />  
    </DarkModeProvider>  
  );  
}
```

#1 Imports the recontextual hook generator from the package

#2 Defines the interface for the context contents

#3 Calls the package function and destructs a provider and a selector hook from the response

#4 Calls the selector hook as before, but we don't define it ourselves this time; the package does that job for us.

#5 Calls the selector hook as before, but we don't define it ourselves this time; the package does that job for us.

#6 Provides the context in the usual way, using the Provider variable returned from the package

EXAMPLE: DARK-MODE-TYPED

This example is in the `ch02/dark-mode-typed` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch02/dark-mode-typed
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: <https://reactlikea.pro/ch02-dark-mode-typed>.

Some of this code may seem a bit complex if TypeScript is new to you. Don't worry. I cover all the details, such as interfaces and strict typing, in chapters 5 and 6.

2.1.5 How useful is the Provider pattern?

The Provider pattern seems to be a minor pattern that may be smart to use for some functionality, but it is a lot more. You can use this single pattern throughout even a large application as the single way to

distribute and organize data and functionality in your entire application.

Your application can have dozens of contexts on many layers working on top of one another, providing global and local functionality to parts or all of your application. You can have your user authorization with the current user information, as well as methods to log in and log out in one context; you can have your application data in a second context, and you can, have data controlling the UI in a third context.

If it's used correctly, this pattern is the single most powerful one in your React quiver because it can apply to almost every application. You'll see this pattern used in several future chapters as we create more complex applications. The Provider pattern is extremely generic and versatile enough to apply to any situation, yet customizable for many constructions.

Some people might argue that instead of using React context with `useContextSelector` to manage complex state throughout your application, it is better to use an established tool such as `redux-toolkit`. Truth be told, `redux-toolkit` uses this same functionality under the hood to provide its magic, so you're getting the same performance with either method. I will discuss other pros and cons of using context versus `redux-toolkit` (and other state management solutions) in chapter 7.

2.2 The Composite pattern

In this section, we delve into the realm of composite components in React, uncovering how they enable the creation of scalable and maintainable user interfaces. You can see a high-level overview of the Composite pattern in figure 2.5.

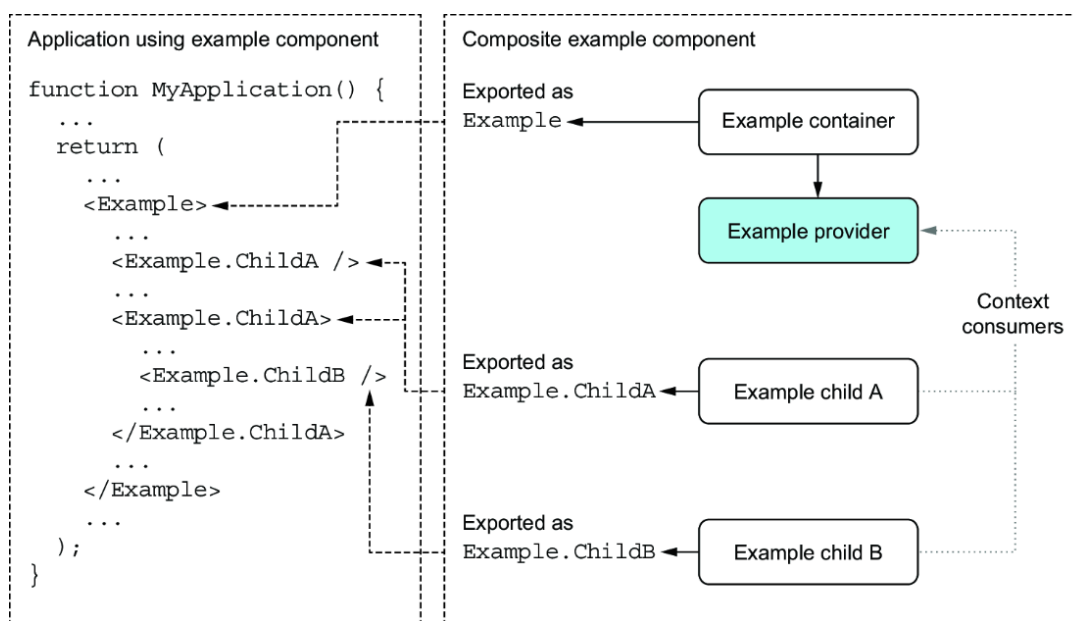


Figure 2.5 A high-level overview of the Composite pattern. From the outside, the components are used as if namespace by the root of the composite component (dashed arrows), but on the inside, they're regular components, communicating via one or more contexts (dotted arrows).

As a key case study, we will examine the evolution of a radio group component. This example will demonstrate the journey from a single component to a composite structure, highlighting the challenges and benefits at each stage. The radio group, starting as a simple UI element and growing in complexity, serves as an ideal illustration of the need for and application of composite components. The following list details the steps we'll go through as we explore the benefits and use cases of this pattern:

- *Simple beginnings with single components*—Initially, the radio group is implemented as a single component, showcasing the simplicity and directness of this approach. Here, the example illustrates the initial ease of development but also sets the stage for potential challenges as complexity increases.
- *The challenges of increasing complexity*—As we add features and requirements to our radio group, the limitations of the single-component approach become apparent. This phase demonstrates how a once-simple component can become cumbersome, leading to a bloated, difficult-to-maintain codebase.
- *The ideal (clean, modular JSX with composite components)*—With the complexities of the radio group established, we explore a mod-

ular composite structure. This approach breaks the radio group into smaller, focused components, demonstrating how it leads to cleaner, more readable code that's easier to maintain.

- *Implementation of composite components* —Finally, we delve into the practical implementation of composite components, using the radio group as our guide. This implementation details how to manage state, props, and context effectively within a composite structure, ensuring that each component remains focused and maintainable. The radio group's transformation exemplifies the benefits of this approach in a real-world scenario.

By the conclusion of this section, you'll have gained a comprehensive understanding of composite components in React and will be equipped with the knowledge to refactor and enhance your own applications. The radio group example will serve as a blueprint for identifying when and how to transition from single to composite components, enhancing both your application's scalability and your development efficiency.

2.2.1 The simple beginnings

You've been tasked with building a breakfast-ordering web application for a restaurant. Customers can order in advance to get their food faster.

There's a chance that this project will evolve, but you've been given no hints about the future, so you don't need to try to anticipate anything. For now, build the minimal application that will meet the current needs. You can see the required output in figure 2.6.

This application is fairly straightforward. We can build it without any custom components by using plain HTML, but that would not teach us anything about React. Let's create a component to generate a list of related radio buttons where we pass in an array of options to be displayed.



Breakfast order form

Meal

- ☐ Small: \$5.99
- ☐ Medium: \$7.99
- ☐ Large: \$9.99

Bread

- ☐ Bagel
- ☐ Roll
- ☐ Croissant

Side

- ☐ Avocado
- ☐ Bacon

Beverage

- ☐ Orange Juice
- ☐ Coffee

Figure 2.6 The desired output of the initial breakfast-ordering form

THE MAIN APPLICATION

This application should be trivial for you to implement. My suggestion is a main application that looks like the following listing.

Listing 2.5 The main app for a simple list of radio groups (excerpt)

```
import { useState } from "react";
import { RadioGroup } from "../RadioGroup";
export default function App() {
  const [data, setData] = useState({
    meal: "",
    bread: "",
    side: "",
    beverage: "",
  });
  const onChange = (name) => (value) =>
    setData({ ...data, [name]: value });
  return (
    <main>
      <h1>Breakfast order form</h1>
      <h2>Meal</h2>
      <RadioGroup
        name="meal"
        options=[
          "Small: $5.99",
          "Medium: $7.99",
          "Large: $9.99",
        ]
        onChange={onChange("meal")}
      />
      <h2>Bread</h2>
      ...
    </main>
  );
}
```

#1 Passes the radio group options as a simple array of strings

THE RADIOGROUP COMPONENT

The implementation of this `RadioGroup` component could look like the following listing.

Listing 2.6 The `RadioGroup` component with a simple interface

```
import { useState } from "react";
export function RadioGroup({ name, options, onChange }) {
  const [selectedValue, setSelectedValue] = useState("");
  const handleChange = (e) => {
    setSelectedValue(e.target.value);
    if (onChange) {
      onChange(e.target.value);
    }
  };
  return (
    <div
      style={{
        display: "flex",
        flexDirection: "column",
        alignItems: "flex-start",
      }}
    >
      {options.map((option) => (
        <label key={option}>
          <input
            type="radio"
            name={name}
            value={option}
            checked={selectedValue === option}      #1
            onChange={handleChange}    #1
          />
          {option}
        </label>
      ))}
    </div>
  );
}
```

#1 Creates a controlled input managed by local state

EXAMPLE: RADIO-SIMPLE

This example is in the `ch02/radio-simple` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch02/radio-simple
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: <https://reactlikea.pro/ch02-radio-simple>.

Note React 19 comes with a new syntax for handling forms by using the `<form action>` property and async action functions. In the context of this example, the new form action function works the same way as a submit handler except that uncontrolled forms are a bit easier to manage. In this example, we have a controlled form, so we would gain nothing by upgrading to React 19 and using a form action.

2.2.2 Complexity increases

The simple version was good. The simple version was bliss. Alas, the world never stays simple. Following in the footprints of the Second Law of Thermodynamics, we might as well define a Second Law of Software Development: *complexity always increases*.

The client loved the breakfast-ordering system but wanted a few changes—a little bit here and there. You can see what the client came up with in figure 2.7.

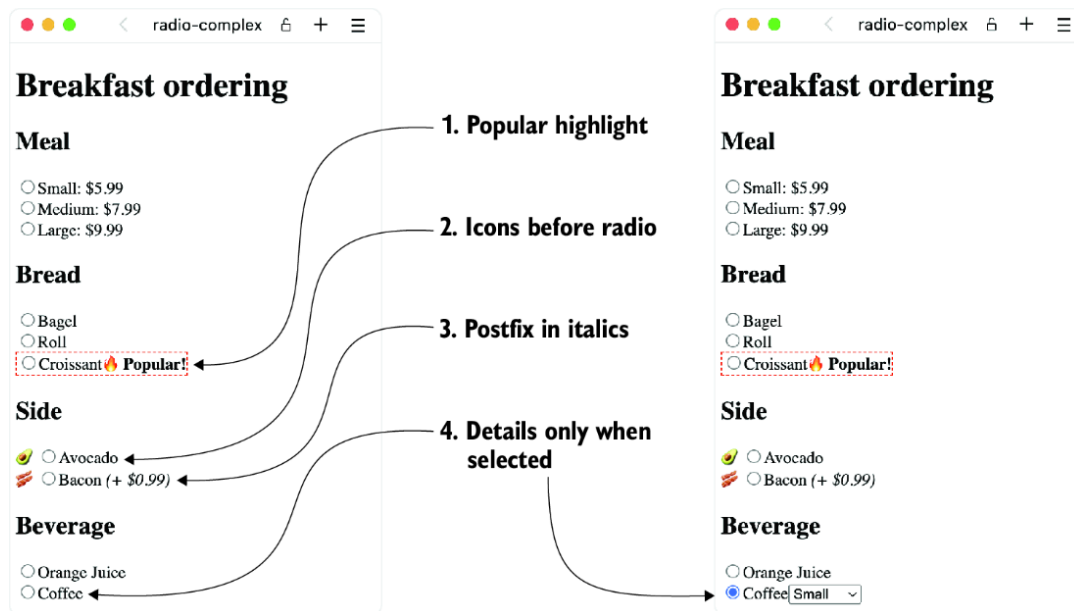


Figure 2.7 The new interface designed by the client. Notice that in the screenshot on right, the coffee beverage is selected. When a selection is made, additional details can be displayed next to the label.

For now, let's not overthink things; we'll expand the existing approach. We still pass in an options array, but now we're going to need more variables per option, so we'll pass in objects with various properties:

```
{
  label: "Bacon",      #1
  value: "bacon",      #2
  isPopular: true,     #3
  icon: "🥓",          #4
  postfix: "(+ $0.99)", #5
  details: <span>Details</span>, #6
}
```

#1 Sets the label to be displayed as the main text after the radio button

#2 Sets the value to be used as the value of the input element

#3 Marks the option as popular (can be omitted)

#4 Sets an optional icon (can be omitted)

#5 Sets an optional postfix (can be omitted)

#6 Adds optional display details if selected (can be omitted)

With that change, the main app file becomes a lot longer but is still relatively straightforward. However, notice that the file is a lot more like regular JavaScript and a lot less JSX now. Most of the component con-

tents are defined in arrays before the JSX is created and returned, as shown in the following listing.

Listing 2.7 Passing option objects to radio groups (excerpt)

```
import { useState } from "react";
import { RadioGroup } from "../RadioGroup";
export default function App() {
  const [data, setData] = useState({
    meal: "",
    bread: "",
    side: "",
    beverage: "",
  });
  const onChange = (name) => (value) =>
    setData({ ...data, [name]: value });
  const meals = [    #1
    { value: "small", label: "Small: $5.99" },
    { value: "medium", label: "Medium: $7.99" },
    { value: "large", label: "Large: $9.99" },
  ];
  const breads = [    #1
    { value: "bagel", label: "Bagel" },
    { value: "roll", label: "Roll" },
    { value: "croissant", label: "Croissant", isPopular: true },
  ];
  const sides = [    #1
    { value: "avocado", label: "Avocado", icon: "🥑" },
    { value: "bacon", label: "Bacon", icon: "🥓" },
  ];
  const beverages = [    #1
    { value: "orangejuice", label: "Orange Juice" },
    {
      value: "coffee",
      label: "Coffee",
      details: (    #2
        <select name="coffee_size">    #2
          <option>Small</option>    #2
          <option>Medium</option>    #2
          <option>Large</option>    #2
        </select>    #2
      ),    #2
    },
  ];
};
```

```
    return (  
      ...  
    );  
  }  
}
```

#1 Defines the four options arrays, where most of the information in this component lies

#2 Adds a details property to the coffee option defined with JSX, which is added inside the options array for beverages

Now the implementation of the radio group has to deal with many more details for each option, and it becomes a bit messy, but it's tolerable, as you can see in the following listing.

Listing 2.8 A radio group with more complex objects

```
import { useState } from "react";
export function RadioGroup({ name, options, onChange }) {
  const [selectedValue, setSelectedValue] = useState("");
  const handleChange = (e) => {
    setSelectedValue(e.target.value);
    if (onChange) {
      onChange(e.target.value);
    }
  };
  return (
    <div
      style={{
        display: "flex",
        flexDirection: "column",
        alignItems: "flex-start",
      }}
    >
      {options.map((option, index) => (
        <label
          key={index}
          style={option.isPopular ? { border: "1px dashed red" } : undefined}
        >
          {option.icon} && <span>{option.icon} </span>
          <input
            type="radio"
            name={name}
            value={option.value}
            checked={selectedValue === option.value}
            onChange={handleChange}
          />
          {option.label}
          {option.postfix && <em> {option.postfix}</em>}
          {selectedValue === option.value && option.details}
          {option.isPopular
            ? <strong>🔥 Popular! </strong>
            : null}
        </label>
      )
      )}
    </div>
  );
}
```

```
    )})  
  </div>  
  );  
}
```

#1 Handles the new isPopular property

#2 Handles the new icon property

#3 Handles the new postfix property

#4 Handles the new details property

EXAMPLE: RADIO-COMPLEX

This example is in the `ch02/radio-complex` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch02/radio-complex
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: <https://reactlikea.pro/ch02-radio-complex>.

Summarizing this new code, we find ourselves navigating a landscape that's rich in features yet teetering on the edge of complexity. Each option, now a small universe of properties and conditions, brings its unique flair to the `RadioGroup`. The once-straightforward component has evolved into a more dynamic entity that's capable of handling icons, popularity indicators, additional charges, and even conditional details.

Although the component handles this increased complexity, it's performing a balancing act. The simplicity of JSX is replaced by the richness of JavaScript logic, where most of the component's character is defined outside the realm of JSX, in the preparatory phase of defining options. This shift from a JSX-dominant structure to a JavaScript-heavy approach illustrates a key aspect of React's flexibility but also underscores the need for careful design choices. As developers, we must constantly weigh the benefits of feature richness against the clarity and maintainability of our code.

In essence, this phase of our journey with the `RadioGroup` component is a microcosm of software development at large. It reflects the constant push and pull between simplicity and complexity, between adding features and maintaining clarity. As we tread further, the need for a more structured approach—one that can handle this complexity gracefully—becomes increasingly apparent. This situation is where composite components, waiting patiently in the wings, come into play, ready to introduce a new paradigm for constructing our React components.

2.2.3 The ideal JSX

What if instead of defining options in JavaScript as an array, we pass in child components for each option to the radio group and then deal with all the complexity of an option inside those components? Then we get back to the ideal of dealing with structure, data, and content in JSX rather than in JavaScript, as we normally do in React. Ideally, we could define the bread group like so:

```
<RadioGroup name="bread" onChange= {onChange("bread")}> #1
  <RadioGroup.Option value="bagel"> #2
    Bagel #2
  </RadioGroup.Option> #2
  <RadioGroup.Option value="roll"> #2
    Roll
  </RadioGroup.Option>
  <RadioGroup.Option value="croissant" isPopular> #3
    Croissant
  </RadioGroup.Option>
</RadioGroup>
```

#1 The radio group itself now takes only two properties.

#2 The options are defined using the subcomponent `RadioGroup.Option`.

#3 Now we can add extra properties for each option as properties of a component rather than an object.

The following listing shows how the entire app would look in this structure.

Listing 2.9 The ideal JSX for the breakfast-ordering component

```
import { useState } from "react";
import { RadioGroup } from "../radiogroup";
export default function App() {
  const [data, setData] = useState({
    meal: "",
    bread: "",
    side: "",
    beverage: "",
  });
  const onChange = (name) => (value) =>
    setData({ ...data, [name]: value });
  return (
    <main>
      <h1>Breakfast ordering</h1>
      <h2>Meal</h2>
      <RadioGroup name="meal" onChange={onChange("meal")}>
        <RadioGroup.Option value="small">
          Small: $5.99
        </RadioGroup.Option>
        <RadioGroup.Option value="medium">
          Medium: $7.99
        </RadioGroup.Option>
        <RadioGroup.Option value="large">
          Large: $9.99
        </RadioGroup.Option>
      </RadioGroup>
      <h2>Bread</h2>
      <RadioGroup name="bread" onChange={onChange("bread")}>
        <RadioGroup.Option value="bagel">Bagel</RadioGroup.Option>
        <RadioGroup.Option value="roll">Roll</RadioGroup.Option>
        <RadioGroup.Option
          value="croissant"
          isPopular #1
        >
          Croissant
        </RadioGroup.Option>
      </RadioGroup>
      <h2>Side</h2>
      <RadioGroup name="side" onChange={onChange("side")}>
        <RadioGroup.Option icon="🥑" value="avocado">
```

```

        Avocado
    </RadioGroup.Option>

    <RadioGroup.Option icon="🥑" value="bacon">
        Bacon <em>(+ $0.99)</em>    #3
    </RadioGroup.Option>
</RadioGroup>
<h2>Beverage</h2>
<RadioGroup name="beverage" onChange={onChange("beverage")}>
    <RadioGroup.Option value="orangejuice">
        Orange Juice
    </RadioGroup.Option>
    <RadioGroup.Option value="coffee">
        Coffee
        <RadioGroup.Details>    #4
            <select name="coffee_size">
                <option>Small</option>
                <option>Medium</option>
                <option>Large</option>
            </select>
        </RadioGroup.Details>
    </RadioGroup.Option>
</RadioGroup>
</main>
);
}

```

#1 Sets the isPopular property directly on the relevant option

#2 Adds an icon to an option to be displayed before the radio input

#3 Defines a label postfix directly as a child of the option component without the need for an extra property

#4 Specifies the details element, visible only when the corresponding option is selected; uses another component exposed by the radio group

This notion of having subcomponents as part of a larger component is the key to the Composite pattern. A radio group is defined not by one component but by two or three components, depending on how complex the group is.

2.2.4 Implementation with composite components

You might be thinking about how you'd implement this example. Two things should be clear: we need to pass information from the radio

group to each option, and we need to pass information from an option to a potential details component inside it.

We have a couple of ways to perform these tasks, but the simplest solution is to use a React context for each information channel. You can see the required information flow in figure 2.8.

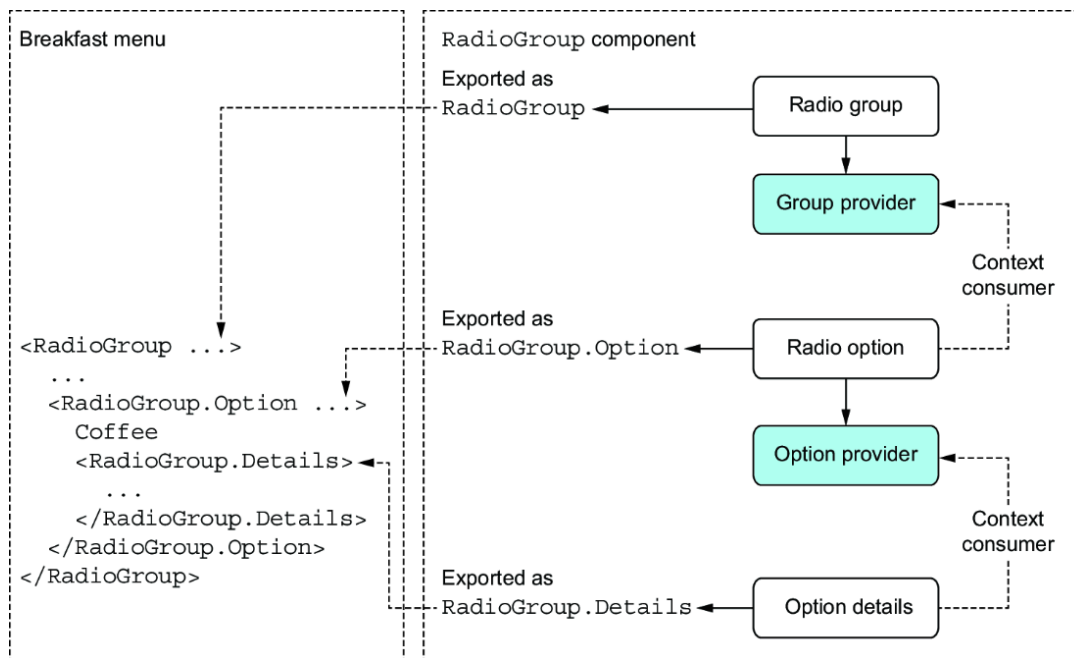


Figure 2.8 We need to pass information from a radio group to an option, including the input name, the currently selected value, and the callback to invoke when the option is selected, which happens through the group provider. We also need to pass some information from the option to the details element, if present—namely, whether it is selected. That happens through the option provider.

Now we're going to implement this iteration of the application in four different files. One file simply defines the contexts to be used by the other three files, which are the individual components. The structure looks like this:

```
/src
  /radiogroup
    contexts.js
    Details.jsx
    index.js
    Option.jsx
    RadioGroup.jsx
  App.jsx
```

We also have `index.js` there, but it simply reexports the main `RadioGroup` component, so the `index.js` file is trivial. Let's start with the contexts in the following listing.

Listing 2.10 Defining the required contexts

```
import { createContext } from "react";
export const RadioGroupContext = createContext();
export const RadioOptionContext = createContext();
```

This file is trivial. We'll use the `RadioGroupContext` to pass information from a radio group to its options and a `RadioOptionContext` to pass information from an option to a potential details element. Next up is the equally trivial `Details` component.

Listing 2.11 The `Details` component

```
import { useContext } from "react";
import { RadioOptionContext } from "../contexts";
export function Details({ children }) {
  const isSelected = useContext(RadioOptionContext);
  return isSelected ? children : null;
}
```

The component simply displays its children if—and only if—its parent radio option is selected. Next is the radio `Option` component, and things get a bit more complex, but this component is still manageable.

Listing 2.12 The Option component

```
import { useContext } from "react";
import { RadioGroupContext, RadioOptionContext } from "../contexts";
export function Option(
  { value, icon, isPopular, children }      #1
) {
  const { name, selectedValue, onChange } =      #2
    useContext(RadioGroupContext);
  const isSelected = selectedValue === value;
  return (
    <label style={isPopular ? { border: "1px dashed red" } : null}>
      {icon}
      <input
        type="radio"
        value={value}
        name={name}
        checked={isSelected}
        onChange={() => onChange(value)}
      />
      <RadioOptionContext.Provider value={isSelected}>      #3
        {children}      #3
      </RadioOptionContext.Provider>      #3

      {isPopular ? <strong> Popular!</strong> : null}
    </label>
  );
}
```

#1 Accepts the four properties required to handle all the variants we need

#2 Reads some important values from the parent radio group

#3 Wraps the children in a context to allow anything in there to figure out whether the option is selected

Finally, we have the main `RadioGroup` component. This component is a lot simpler now because it doesn't have any options; it mostly accepts and passes through the children it receives.

Listing 2.13 The `RadioGroup` component and its composites

```
import { useState } from "react";
import { RadioGroupContext } from "../contexts";
import { Option } from "../Option";
import { Details } from "../Details";
export function RadioGroup({ children, name, onChange }) {
  const [selectedValue, setSelectedValue] = useState("");
  const handleChange = (value) => {
    setSelectedValue(value);
    if (onChange) {
      onChange(value);
    }
  };
  const contextValue = {
    name,
    selectedValue,
    onChange: handleChange,
  };
  return (
    <div
      style={{
        display: "flex",
        flexDirection: "column",
        alignItems: "flex-start",
      }}
    >
      <RadioGroupContext.Provider #1
        value={contextValue} #1
      > #1
        {children} #1
      </RadioGroupContext.Provider> #1
    </div>
  );
}
RadioGroup.Option = Option; #2
RadioGroup.Details = Details; #2
```

#1 Wraps the children in the proper context

#2 Reexports the constituent components as properties of the main component, so we need only a single import to use the radio group

EXAMPLE: RADIO-COMPOSITE

This example is in the `ch02/radio-composite` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch02/radio-composite
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: <https://reactlikea.pro/ch02-radio-composite>.

With this code, we have all the components defined, and individually, they're a lot simpler than in the previous complex example, in which a single component handled all the responsibilities. The beauty of the Composite pattern is that it allows for a harmonious symphony of distinct functionalities, each encapsulated in its own component yet all working together seamlessly. This modular approach not only makes the codebase more intuitive and manageable but also enhances its extendability. With each component handling a specific aspect of the radio group's behavior, we've moved from a monolithic, rigid structure to a flexible, scalable architecture.

In this new paradigm, the `RadioGroup` acts as the conductor, orchestrating the flow of data and events through contexts, while each `Option` and `Details` component plays its part with focused responsibility. The `RadioGroupContext` elegantly manages the shared state and behavior among the options, ensuring a cohesive user experience. The `RadioOptionContext`, on the other hand, provides a direct communication channel to the `Details` component, allowing for dynamic content rendering based on the selection state.

The result is a `RadioGroup` component that is not only functionally rich but also a pleasure to work with and extend. It's a testament to the power of breaking complex interfaces into smaller, more manageable pieces, each with a clear purpose. This approach not only simplifies development and maintenance but also opens the door to more creative and complex UI designs while keeping the codebase clean and

approachable. The journey from clutter and confusion to clarity and elegance showcases the transformative power of the Composite pattern in React development.

2.2.5 How useful is the Composite pattern?

The Composite pattern may initially appear to be a mere structural convenience for organizing components, but its utility in React development runs much deeper. This pattern is not just a tool for creating cleaner code; it's also a fundamental strategy for building scalable and maintainable applications. In a large-scale application, the Composite pattern can be the key to managing complex component hierarchies, allowing for a modular and flexible architecture.

Imagine your application as a collection of interrelated components, each serving a specific purpose. With the Composite pattern, you can construct intricate UIs by combining these components in various configurations, each component acting as a building block. A complex form could be composed of various input components, validation messages, and control buttons, each defined separately but all working together as a cohesive unit.

Some developers might prefer using direct parent–child relationships or other state management techniques for organizing component structures. But the Composite pattern offers a level of abstraction and flexibility that these methods often lack. By decoupling child components from their parent, the Composite pattern allows for greater reuse and easier modification of individual components.

This pattern excels when UI components need to be reused and recombined in various ways, making it an invaluable asset in your React toolkit. Understanding when and how to employ the Composite pattern effectively is crucial for any React developer who aims to build sophisticated, robust web applications.

2.3 The Summary pattern

All right, let's dive into our third and last React design pattern:

Summary. Don't let its simple appearance fool you; this pattern is a game changer. It makes React components sleek and efficient, especially the part above the JSX return. Think of the Summary pattern as a neat trick for tidying your code by packing logic into custom hooks. Using this pattern is like giving your components a makeover for a more streamlined, sophisticated look. As a bonus, your fellow devs will thank you for making collaboration a breeze.

Now comes the fun part. We're going to walk through two examples that show off how handy the Summary pattern is. First, we'll see how a single custom hook can do wonders in simplifying a component, making it easy like Sunday morning. This example shows how one well-crafted hook can make your component look and function better.

Next, we'll jump into a trickier scenario in which two or three custom hooks come into play. The Summary pattern shows its true colors in handling complex situations with style. Splitting the logic into multiple hooks not only organizes your component better but also opens a world of reusability and flexibility.

Figure 2.9 is your visual guide to the way the Summary pattern transforms the traditional component structure into something more refined and efficient. The pattern is like before-and-after photos for your React components. So let's get on with it and see how this seemingly simple pattern can have a big effect on your React development.

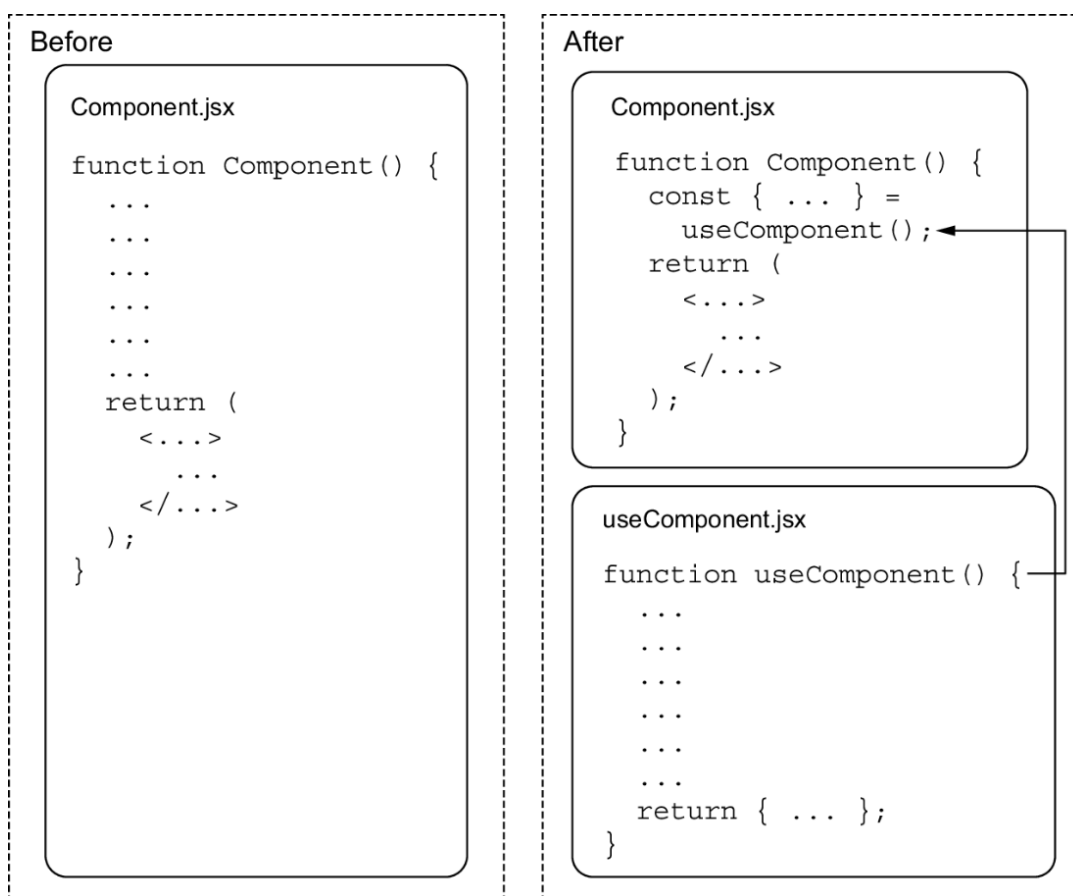


Figure 2.9 The essence of the Summary pattern captured in a simple diagram. Before, the component consisted of a bunch of code before the JSX and then a bunch of JSX. After, the component is a few lines before the JSX, and all the remaining code has been put in a custom hook specific to that component.

2.3.1 A single custom hook

We will start with a component that should be familiar to you: the `RadioGroup` component from listing 2.13 a few pages back. First, let's zoom in on the part of the component that comes before the return statement:

```

const [selectedValue, setSelectedValue] = useState("");
const handleChange = (value) => {
  setSelectedValue(value);
  if (onChange) {
    onChange(value);
  }
};
const contextValue = {
  name,
  selectedValue,
  onChange: handleChange,
};

```

This snippet isn't a lot of code, but it could be even less. The only value we need for the JSX is the `contextValue` at the end, so if we were to create a custom hook, all we'd need from it would be that object. So we could rewrite the code as

```

const contextValue = useContextValue({ name, onChange });

```

Note that we need to pass in both the `name` and `onChange` properties, as they're used by the context. And yes, we can easily create such a custom hook like so:

```

import { useState } from "react";
export function useContextValue({ name, onChange }) {
  const [selectedValue, setSelectedValue] = useState("");
  const handleChange = (value) => {
    setSelectedValue(value);
    if (onChange) {
      onChange(value);
    }
  };
  return {
    name,
    selectedValue,
    onChange: handleChange,
  };
});

```

We've essentially moved these lines to a new function. That's it. But the result is a much cleaner `RadioGroup` component, as you can see in the following listing.

Listing 2.14 The `RadioGroup` component summarized

```
import { RadioGroupContext } from "../contexts";
import { Option } from "../Option";
import { Details } from "../Details";
import { useContextValue } from "../useContextValue";
export function RadioGroup({ children, name, onChange }) {
  const contextValue = #1
  useContextValue({ name, onChange }); #1
  return (
    <div
      style={{
        display: "flex",
        flexDirection: "column",
        alignItems: "flex-start",
      }}
    >
      <RadioGroupContext.Provider value={contextValue}>
        {children}
      </RadioGroupContext.Provider>
    </div>
  );
}
RadioGroup.Option = Option;
RadioGroup.Details = Details;
```

#1 Prepares for the JSX return in a single line invoking the new custom hook

EXAMPLE: RADIO-SUMMARY

This example is in the `ch02/radio-summary` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch02/radio-summary
```

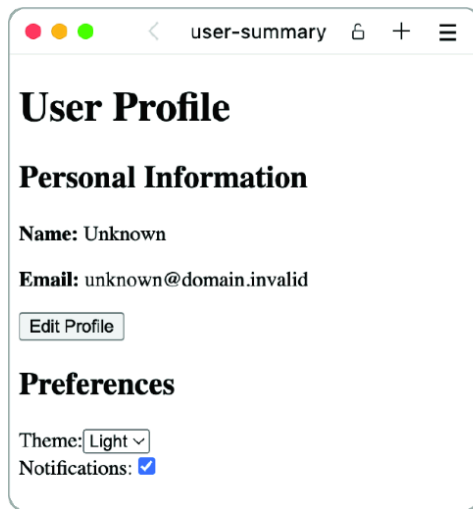
Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: <https://reactlikea.pro/ch02-radio-summary>.

That's it. This example seems deceptively simple, and it is. But your code will look so much more impressive with this one small change. Trust me!

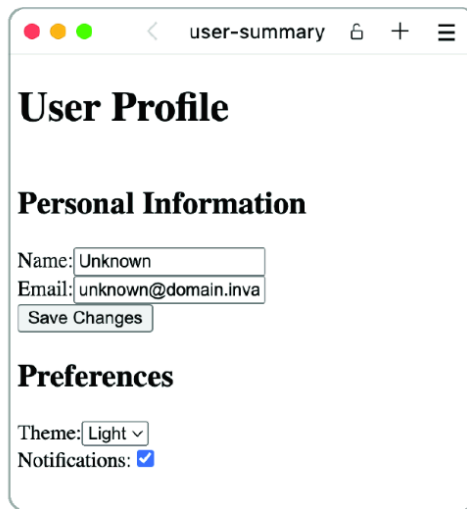
Note that we removed more lines than just the few inside the component. We also got rid of an import because we no longer need `useState` in this file. Yes, we did add a new import for the custom hook, but often, we can remove more than one import.

2.3.2 Better results with more complexity

Let's look at a more complex component with many more dependencies and more regular hooks. This time, we're building a user profile component; the user can see and edit their user data as well as update their preferences (such as theme and notifications). The result looks like figure 2.10. (Please imagine that it looks pretty, with some professional styles.)



The image shows a web browser window with the address bar displaying 'user-summary'. The page has a title 'User Profile' and a section 'Personal Information'. Under 'Personal Information', it shows 'Name: Unknown' and 'Email: unknown@domain.invalid'. There is an 'Edit Profile' button. Below this is a 'Preferences' section with 'Theme: Light' (a dropdown menu) and 'Notifications: ☒'.



The image shows the same web browser window but in edit mode. The 'Personal Information' section now has input fields for 'Name' (containing 'Unknown') and 'Email' (containing 'unknown@domain.inva'). There is a 'Save Changes' button. The 'Preferences' section remains the same with 'Theme: Light' and 'Notifications: ☒'.

Figure 2.10 The user profile page in view mode and edit mode, respectively

To implement this page, we have a hook to access our API functionality from a shared place, called `useAPI`. We also have components to render the user data, user data edit form, and user preferences (equally obviously named). So let's see what this component looks like in a common setup. We put all the basic and custom hooks used directly in the component itself.

Listing 2.15 User profile before summarizing

```
import { useState, useEffect } from "react";
import { UserDataForm } from "../UserDataForm";
import { UserDetails } from "../UserDetails";
import { UserPreferences } from "../UserPreferences";
import { useAPI } from "../useAPI";
export function LongUserProfile({ userId }) {
  const [userData, setUserData] = useState(null); #1
  const [editMode, setEditMode] = useState(false); #1
  const [userPreferences, setUserPreferences] = #1
    useState({ #1
      theme: "light", #1
      notifications: true, #1
    }); #1
  // Fetching user data
  const api = useAPI(); #2
  useEffect(() => { #3
    api
      .fetchUser(userId)
      .then((response) => response.json())
      .then((data) => setUserData(data));
  }, [api, userId]);
  // Initializing user preferences
  useEffect(() => {
    const storedPreferences = localStorage.getItem("userPreferences");
    if (storedPreferences) {
      setUserPreferences(JSON.parse(storedPreferences));
    }
  }, []);
  // Updating user preferences
  useEffect(() => { #4
    localStorage.setItem(
      "userPreferences",
      JSON.stringify(userPreferences)
    );
  }, [userPreferences]);
  // Toggle edit mode
  const toggleEditMode = #5
    () => setEditMode(!editMode); #5
  // Update preferences #5
  const updatePreferences = #5
```

```

    (newPreferences) =>      #5
      setUserPreferences(newPreferences);    #5
    if (!userData) return <div>Loading...</div>;
    return (
      <div>
        <h1>User Profile</h1>
        {editMode ? (
          <UserDataForm
            userData={userData}
            onChange={setUserData}    #6
          />
        ) : (
          <UserDetails userData={userData} />
        )}
        <button onClick={toggleEditMode}>    #6
          {editMode ? "Save Changes" : "Edit Profile"}
        </button>
        <UserPreferences
          preferences={userPreferences}
          onPreferencesChange={updatePreferences}    #6
        />
      </div>
    );
  }
}

```

#1 Creates three stateful values using useState

#2 Accesses the API using a custom hook

#3 Registers three effects using useEffect

#4 Registers three effects using useEffect

#5 Creates callbacks to be used in the JSX

#6 Uses callbacks created above the JSX to add interactivity where required

Now let's move all the logic before the (first) return to a custom hook.

Listing 2.16 User profile custom hook

```
import { useEffect, useState } from "react";
import { useAPI } from "../useAPI";

function useFetchUserData(userId) {      #1
  const [userData, setUserData] = useState(null);
  const api = useAPI();
  useEffect(() => {
    // API call to fetch user data
    api
      .fetchUser(userId)
      .then((response) => response.json())
      .then((data) => setUserData(data));
  }, [api, userId]);
  return userData;
}

export function useUserProfile(userId) {    #2
  const [editMode, setEditMode] = useState(false);
  const [userPreferences, setUserPreferences] = useState({
    theme: "light",
    notifications: true,
  });
  const userData = useFetchUserData(userId);
  useEffect(() => {
    const storedPreferences = localStorage.getItem("userPreferences");
    if (storedPreferences) {
      setUserPreferences(JSON.parse(storedPreferences));
    }
  }, []);
  useEffect(() => {
    localStorage.setItem(
      "userPreferences",
      JSON.stringify(userPreferences)
    );
  }, [userPreferences]);
  const toggleEditMode = () => setEditMode(!editMode);
  const updatePreferences = (newPreferences) =>
    setUserPreferences(newPreferences);
  return {      #3
    userData,    #3
    editMode,    #3
    userPreferences,    #3
  };
}
```

```
toggleEditMode, #3  
updatePreferences, #3  
};  
}
```

#1 Creates a small custom hook for part of the logic

#2 Creates the main custom hook that captures all the logic required

#3 Returns all the values needed by the component for display

With that task done, our base component looks like the following listing.

Listing 2.17 User profile after summarization

```
import { useUserProfile } from "./useUserProfile";
import { UserDataForm } from "./UserDataForm";
import { UserDetails } from "./UserDetails";
import { UserPreferences } from "./UserPreferences";
export function CompactUserProfile({ userId }) {
  const {      #1
    userData,   #1
    editMode,   #1
    userPreferences,  #1
    toggleEditMode,  #1
    updatePreferences,  #1
  } = useUserProfile(userId);  #1
  if (!userData) return <div>Loading...</div>;
  return (
    <div>
      <h1>User Profile</h1>
      {editMode ? (
        <UserDataForm
          userData={userData}
          onSave={toggleEditMode}
        />
      ) : (
        <UserDetails userData={userData} />
      )}
      <button onClick={toggleEditMode}>
        {editMode ? "Save Changes" : "Edit Profile"}
      </button>
      <UserPreferences
        preferences={userPreferences}
        onPreferencesChange={updatePreferences}
      />
    </div>
  );
}
```

#1 Destructs all the values from the custom hook. Even though this destructuring takes up many lines, we still save a ton overall.

EXAMPLE: USER-SUMMARY

This example is in the `ch02/user-summary` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch02/user-summary
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: <https://reactlikea.pro/ch02-user-summary>.

That's a much shorter component! We went from 58 to 34 lines (ignoring empty lines) in the component file. We did add a second file, so the total line count has increased, but each file is short and condensed and has a simple, single purpose—a huge improvement and a much cleaner component. The best part is that if you're an experienced developer, you might not even have to look inside the `useUserProfile` hook because what it does is obvious based on the return values and how they're used—an instant LGTM (Looks Good To Me) in the pull-request code review!

Note The example in listings 2.16 and 2.17 would be even cleaner if we also adopted the Provider pattern and wrapped all the child components in a provider. But I'll leave that task as an exercise for you.

2.3.3 Multiple hooks required

Let's create a simple task manager this time around. You can see it at different stages in figure 2.11. Let's implement it in a single big component, as shown in listing 2.18.

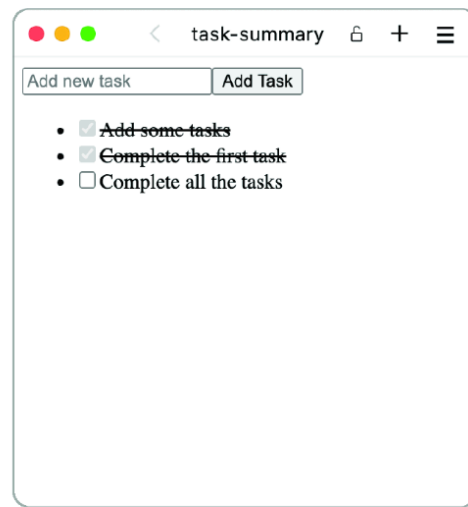
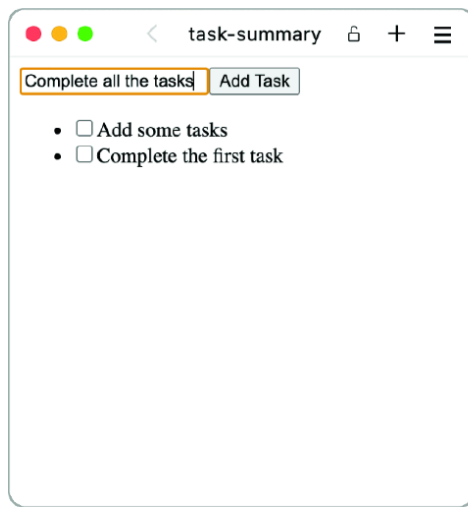


Figure 2.11 The task manager in action. We can add, see, and complete tasks. What else would we ever want to do?

Listing 2.18 Task manager before summarization (excerpt)

```
import { useState, useCallback } from "react";
export function LongTaskManager() {
  const [newTask, setNewTask] = useState("");
  const [tasks, setTasks] = useState([]);
  const handleNewTaskChange = (e) => {
    setNewTask(e.target.value);
  };
  const addTask = (task) => {
    if (task.trim() !== "") {
      setTasks((prevTasks) => [
        ...prevTasks,
        { task, completed: false },
      ]);
    }
  };
  const toggleTaskCompletion = useCallback(    #1
    (index) => {
      setTasks((prevTasks) =>
        prevTasks.map((task, i) =>
          i === index ? { ...task, completed: !task.completed } : task
        )
      );
    },
    [],
  );
  const handleSubmit = (e) => {
    e.preventDefault();
    addTask(newTask);
    setNewTask("");
  };
  return (
    ...    #2
  );
}
```

#1 Wraps a callback in a memoization hook. This code doesn't do anything in this instance, as we'll discuss in chapter 3, but it's included for the example.

#2 The returned JSX is quite straightforward but skipped here for brevity. You can see it in the attached task-summary example or in listing 2.21.

Nothing here is surprising. You should be able to write this component, and most developers would do it something like listing 2.18. But we can do better: we have 20-plus lines of logic before the return that can be reduced significantly for improved overview!

Although we could extract the entire logic into a single hook, and nothing is stopping us, the logic kind of splits into two groups of code. One group deals with the task list and with adding and toggling tasks inside it; the other deals with the add-new-task form and its internal logic required to update the form's state and handle the submit. These two nonoverlapping responsibilities make it a prime candidate for two separate hooks, though they won't be completely separate because the form-handling hook needs to know about the `addTask` function from the task-list hook. The following listing shows how we could move the task-list logic to a separate hook.

Listing 2.19 Task-list custom hook

```
import { useCallback, useState } from "react";
export function useTaskList() {
  const [tasks, setTasks] = useState([]);
  const addTask = (task) => {
    if (task.trim() !== "") {
      setTasks((prevTasks) => [
        ...prevTasks,
        { task, completed: false },
      ]);
    }
  };
  const toggleTaskCompletion = useCallback((index) => {
    setTasks((prevTasks) =>
      prevTasks.map((task, i) =>
        i === index ? { ...task, completed: !task.completed } : task
      )
    );
  }, []);
  return { tasks, addTask, toggleTaskCompletion };    #1
}
```

#1 Returns the three properties we need

Next is the `useNewTask` hook, which takes the `addTask` function from the `useTaskList` hook as an argument.

Listing 2.20 New-task custom hook

```
import { useState } from "react";
export function useNewTaskInput(addTask) {      #1
  const [newTask, setNewTask] = useState("");
  const handleNewTaskChange = (e) => {
    setNewTask(e.target.value);
  };
  const handleSubmit = (e) => {
    e.preventDefault();
    addTask(newTask);
    setNewTask(""); // Reset input after adding task
  };
  return { #2
    newTask, #2
    handleNewTaskChange, #2
    handleSubmit #2
  };      #2
}
```

#1 Accepts the `addTask` function to be passed in by the main component

#2 Returns the three properties we need to handle adding tasks

The following listing shows the new compact task manager with the logic summarized into two hooks.

Listing 2.21 The optimized task-list manager

```
import { useNewTaskInput } from "./useNewTask";
import { useTaskList } from "./useTaskList";
export function CompactTaskManager() {
  const { tasks, addTask, toggleTaskCompletion } = #1
    useTaskList(); #1
  const { #2
    newTask, #2
    handleNewTaskChange, #2
    handleSubmit #2
  } = useNewTaskInput(addTask); #2
  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          value={newTask}
          onChange={handleNewTaskChange}
          placeholder="Add new task"
        />
        <button type="submit">Add Task</button>
      </form>
      <ul>
        {tasks.map((task, index) => (
          <li
            key={index}
            style={{
              textDecoration: task.completed ? "line-through" : "none",
            }}
          >
            <input
              type="checkbox"
              checked={task.completed}
              onChange={() => toggleTaskCompletion(index)}
              disabled={task.completed}
            />
            {task.task}
          </li>
        ))}
      </ul>
    </div>
```

```
);  
}
```

#1 Invokes the task-list custom hook first

#2 Invokes the add-task custom hook second, as it needs a value from the first hook

EXAMPLE: TASK-SUMMARY

This example is in the `ch02/task-summary` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch02/task-summary
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: <https://reactlikea.pro/ch02-task-summary>.

The choice between using a single hook or multiple hooks is completely arbitrary, of course, and up to the developer in charge. In this instance, a single hook might have been okay, but I prefer a two-hook version for this example.

2.3.4 How useful is the Summary pattern?

In conclusion, the Summary pattern stands out for its ability to streamline and organize React components. After exploring its application through single or multiple custom hooks, we see that this pattern is a key player in enhancing code clarity and maintainability.

The strength of the Summary pattern lies in its versatility and scalability. Whether we're simplifying a component with a single hook or orchestrating several hooks for more complex scenarios, it offers a tailored approach to managing component logic. This pattern not only cleans up code but also fosters a mindset of reusability, making it invaluable for larger projects in which consistency is crucial.

In essence, the Summary pattern may be understated, but its contribution to creating efficient, manageable React applications is undeniable.

It exemplifies the principle that effective solutions often lie in simplicity and thoughtful organization.

Summary

- Effective software development mirrors core principles of architecture, emphasizing stability, structure, and use of appropriate patterns.
- The Provider pattern in React centralizes and simplifies data management, making it indispensable for global state control in complex applications.
- By using the Provider pattern, React developers can pass data and functions efficiently through component hierarchies, enhancing organization and functionality.
- The Composite pattern in React offers a strategic approach to managing complex UI structures, enabling modular and flexible component architecture.
- Embracing the Composite pattern allows for dynamic and complex UI designs, fostering component reusability and structural integrity in React projects.
- The Summary pattern in React focuses on reducing code clutter by abstracting logic into custom hooks, leading to cleaner, more maintainable components.
- Implementing the Summary pattern enhances the readability and scalability of React components, contributing to efficient and collaborative coding practices.
- In React development, adhering to foundational design patterns such as Provider, Composite, and Summary ensures the creation of robust and innovative applications.
- Mastering these React design patterns gives developers the tools to build diverse and stable digital structures, much like in physical construction.