6 Mastering TypeScript with React

This chapter covers

- Mastering advanced TypeScript techniques specific to React
- Typing React hooks
- Designing complex component patterns with generics

In this chapter, we delve deeper into the synergy between TypeScript, React components, and the broader application landscape. Imagine your React components as the tenants in a larger property; the application as a whole acts as the landlord. TypeScript takes center stage as the binding contract between these two entities, ensuring clarity, security, and smooth interactions. This chapter marks a crucial step in your journey to mastering the art of writing reliable, scalable, and maintainable React applications by enhancing the tenant–landlord relationship with TypeScript as the pivotal contract.

TypeScript brings clarity to the responsibilities and obligations of each party, making the entire development process more efficient and reliable. As you progress through this chapter, you'll discover how TypeScript reinforces this contract, making your React applications more robust and resilient. The chapter covers the following topics to round off your TypeScript edification (for now):

- Exploring the intricacies of typing React hooks to ensure that your components interact seamlessly with the world beyond
- Uncovering the power of well-typed React hooks as exceptional building blocks for shared functionality in your applications
- Discovering the practical application of generics in building a versatile and reusable component for data pagination

- Understanding the potential challenges and complexities of TypeScript integration and how to mitigate them

By the end of this chapter, you'll have not only a solid grasp of using TypeScript with React but also the knowledge to apply these concepts in various professional web development projects. Let's explore how TypeScript further elevates your React skills, making you a master of this powerful combination, including the exceptional potential of well-typed hooks as building blocks for shared functionality within your applications.

Note The source code for the examples in this chapter is available at https://reactlikea.pro/ch06.

6.1 Using React hooks in TypeScript

Hooks play a central role in modern React development, which still holds true when TypeScript is introduced. Nevertheless, certain considerations apply to working with hooks in a typed environment; keeping type definitions clear is vital, and at times, some guidance is needed for the TypeScript interpreter.

Of the various hooks, two of the most fundamental—`useState` and `useRef`—can be surprisingly complex to type correctly. This conundrum is an interesting one because these hooks are among the most commonly used.

This section begins by addressing these two hooks, providing insights into typing them effectively. Then we'll explore React context, which is not overly complex but require some attention. Finally, we'll discuss effects, reducers, and memoization hooks. This list skips a few of the React hooks, but they aren't relevant to this discussion because they're extremely simple typewise or too advanced functionally to cover in this book.

6.1.1 Typing useState

The `useState` hook needs one piece of information for typing: the type of the state stored by using the hook. Sometimes, this information is trivial; at other times, the type of the data stored varies over time, and we need to inform the hook about all the potential types on initialization so we can use it in a typesafe manner.

INFERRING TYPE FROM INITIALIZER

Suppose that you're building an appointment-booking website. When an appointment arrives, the recipient can reject or accept the appointment and provide a message.

This task calls for a small component with two states. One state, `isAccepted`, is a Boolean that holds whether the appointment was accepted. Let's set it to `true` by default, as we assume that most recipients will accept appointments as booked. The second state, `message`, is a string with an optional reason or additional details. Let's set it to an empty string by default, indicating that no additional details are present. In pure JavaScript, here's how we would create these two state values in React:

```
const [isAccepted, setIsAccepted] = useState(true);
const [message, setMessage] = useState("");
```

In TypeScript, we'll do the same thing. This code works as well and as desired in TypeScript—nothing of note to discuss.

Imagine a slightly different scenario, however. This time, we won't assume anything about the response of the recipient, so we'll default the `isAccepted` value to `null`, indicating that no selection has been made yet. In plain JavaScript, we'd initialize the value this way:

```
const [isAccepted, setIsAccepted] = useState(null);
```

But if we do the same in TypeScript, we'll run into problems later. If we update the value by calling `setIsAccepted(true)`, we get this TypeScript error:

```
Argument of type 'true' is not assignable to parameter of type
'SetStateAction<null>'.
```

The error message says that we invoked the setter function (which is typed internally in React with the interface `SetStateAction`) with a value that did not match the expected one. `null` was expected, and we passed in a Boolean. We created a stateful variable that can hold only the value `null`.

TypeScript infers the value of the state contents from the initial value, and when we provided `boolean` as the initial value, the type was inferred as Boolean. But now that we're providing `null`, the type is inferred to be `null`, and only a single value has that type: `null` itself.

That result is no good, of course. We intended the type of the state contents to be `null` or `boolean`. We inform TypeScript of that by providing a type argument to `useState`:

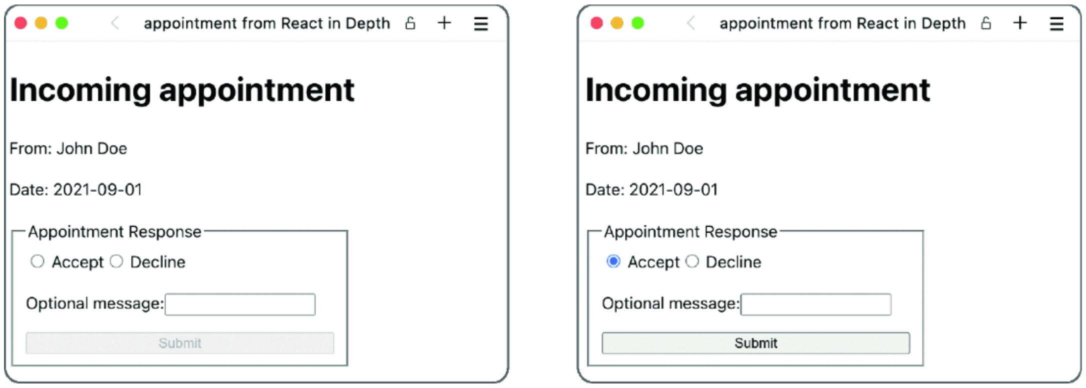
```
const [isAccepted, setIsAccepted] = useState<null | boolean>(null);
```

Now our booking component works. The radio buttons for selecting whether the appointment is accepted or rejected are initially unselected but can update as the user interacts with the component. You can see the full code for this component in the following listing and the output in figure 6.1.

Listing 6.1 The appointment-response component

```
import { useState } from "react";
interface AppointmentResponseProps {
  onSubmit:
    (data: { isAccepted: boolean; message: string }) => void;
}
export function AppointmentResponse({
  onSubmit,
}: AppointmentResponseProps) {
  const [isAccepted, setIsAccepted] =     #1
    useState<null | boolean>(null);    #1
  const [message, setMessage] = useState("");    #2
  const canSubmit = typeof isAccepted === "boolean";
  return (
    <div>
      <fieldset
        style={{
          display: "flex",
          flexDirection: "column",
          gap: "1em",
          width: "300px",
        }}
      >
        <legend>Appointment Response</legend>
        <div>
          <label>
            <input
              type="radio"
              name="is_accepted"
              checked={isAccepted === true}
              onChange={
                () => setIsAccepted(true)     #3
              }  #3
            />{" "}  #3
            Accept  #3
          </label>  #3
          <label>  #3
            <input  #3
              type="radio"  #3
              name="is_accepted"   #3
              checked={isAccepted === false}   #3
              onChange={   #3
                () => setIsAccepted(false)     #3
              }   #3
            />{" "}   #3
            Decline   #3
          </label>   #3
        </div>   #3
        <label>   #3
          Optional message:   #3
          <input   #3
            type="text"   #3
            value={message}   #3
            onChange={   #3
              ({ target: { value } }) =>  #3
                setMessage(value)     #3
            }
          />
        </label>
        <button
          disabled={!canSubmit}
          onClick={() => canSubmit && onSubmit({ isAccepted, message })}
        >
          Submit
        </button>
      </fieldset>
    </div>
  );
}
```

#1 Creates the isAccepting state value using a type argument to allow both null and booleans to be passed in
#2 Creates the message state value without a type argument because we'll pass in only strings anyway
#3 Updates the state values as normal


appointment from React in Depth

Incoming appointment

From: John Doe

Date: 2021-09-01

Appointment Response

Accept Decline

Optional message:

appointment from React in Depth

Incoming appointment

From: John Doe

Date: 2021-09-01

Appointment Response

[✓ Accept] Decline

Optional message:

Figure 6.1 This appointment-response component is initially undecided (left) but can be updated as accepted or rejected (right). This component is powered by the state value, for which `null`, `true`, and `false` are the allowed values.

EXAMPLE: APPOINTMENT

This example is in the `appointment` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch06/appointment
```

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/ch06-appointment.

Note that instead of using a type argument, we could have cast the initializer to the desired type, in this instance to `null | boolean`:

```
const [isAccepted, setIsAccepted] = useState(null as null | boolean);
```

That code looks weird and feels wrong, so you probably won't see a lot of people using it, but it would work.

TYPING ARRAYS IN USESTATE

A related problem happens when you have arrays or objects in your state. Suppose that we're creating a small component, `TagForm`, to add tags to some object. This component might be used in a blog system to allow users to add tags to blog posts.

This component allows the user to type a tag and press Enter; then the tag is added to an internal list of tags and the input is cleared. All the current tags are displayed below the input.

For this task, we need two state values. One value is the value of the input, `newTag`, which is a string we'll initialize to the empty string. The other value is the list of tags, `tags`, an array of strings that we'll initialize to an empty array:

```
const [newTag, setNewTag] = useState("");
const [tags, setTags] = useState([]);
```

Again, we'll run into problems when we try to update this list of tags. The obvious update is

```
setTags((oldTags) => [...oldTags, newTag]);
```

but it results in this TypeScript error:

```
Argument of type '(oldTags: never[]) => string[]' is not assignable to
parameter of type 'SetStateAction<never[]>'.
  Type '(oldTags: never[]) => string[]' is not assignable to type
  '(prevState: never[]) => never[]'.
    Type 'string[]' is not assignable to type 'never[]'.
      Type 'string' is not assignable to type 'never'
```

As when we're creating a regular variable initialized to an empty array, we need to specify what the array can contain. Otherwise, TypeScript assumes that the array can contain nothing (which is an inane default, but it makes as much sense as any other assumption). The obvious solution is a type argument to `useState`:

```
const [tags, setTags] = useState<string[]>([]);
```

You can see the full source code for this application with a tag form in the `tag-form` example.

EXAMPLE: TAG-FORM

This example is in the `tag-form` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch06/tag-form
```

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/ch06-tag-form**.

TYPING OBJECTS IN USESTATE

For objects, we do the same thing. Suppose that we want to create a map of which users are hidden from a given view. For this task, we have a state variable with a map from user id (which is a string) to a Boolean indicating whether the user is hidden.

Again, we cannot initialize the state to an empty object and assume that TypeScript can guess what we want to put there. We need to enter a type argument, such as one using the `Record` interface:

```
const [userVisibility, setUserVisibility] =
  useState<Record<string, boolean>>({});
```

This example is a common way of typing object maps, so you'll see it often.

6.1.2 Typing useRef

When using references in your components, you often run into the same problems as for states. TypeScript automatically infers the contents of the reference from the initializer, but if the initializer does not reflect the full picture of what you want to put there, you run into problems. In common cases, you deal with the problem by providing a type argument with the full picture:

```
const stringOrNullRef = useRef<string | null>(null);
const userArrayRef = useRef<User[]>([]);
```

This code works pretty much as expected if you do what you did for `useState` in section 6.1.1. But `useRef` has some surprises. We can type the first example slightly more simply:

```
const stringOrNullRef = useRef<string>(null);
```

`useRef` has a type overload for the specific case in which the type argument doesn't include `null` but the initializer is `null`. If that's the case, `useRef` expands the type argument to allow `null` implicitly.

But the catch is that the latter definition is a nonmutable reference. Nonmutable references can be passed on to elements, so something like the following would work:

```
const inputRef = useRef<HTMLInputElement>(null);
...
return <input ref={inputRef} />;
```

This example is a common use case of a special overload of `useRef`. We can leave `|` `null` out of the type argument and initialize the `ref` with `null`, and everything works as expected. But if we're creating references for mutable state and want to update that state directly, perhaps in an effect or a callback, this approach doesn't work.

Suppose that we want to create a component that remembers the position of the cursor while it's inside the component in a reference and clears that reference when the mouse leaves. We could implement that component as follows:

```
function MouseTracker() {
  const position = useRef<{ x: number; y: number } | null>(null);
  const onMouseLeave = () => {
    position.current = null;
  };
  const onMouseMove = (evt: MouseEvent) => {
    position.current = { x: evt.clientX, y: evt.clientY };
  };
  return (
    <div onMouseLeave={onMouseLeave} onMouseMove={onMouseMove}>
      ...
```

Notice that we initialize the reference with an explicit `| null` in the type argument. If we left that bit out, the example would not work—not because `null` wasn't allowed but because the reference would be immutable. No direct updates would be allowed, and we'd get this error:

```
Cannot assign to 'current' because it is a read-only property.
```

It's best to be explicit about the possible values of a mutable reference in the type argument to prevent this weird behavior.

Note In React 19, all references are mutable, so this error will no longer happen. You could safely skip `|` `null` in the preceding snippet, and the code would run without errors or warnings.

6.1.3 Typing contexts and useContext

Simple contexts are simple to type. Complex contexts are complex to type. If you use the basic context from React itself (not a selector-based one, as in chapter 2), you'll probably use the context only for fairly simple values.

If we have a context that stores whether we're in dark mode on the website, the context value is a Boolean. For such a simple example, there's no difference between using TypeScript and plain JavaScript, as the types are easily inferred:

```
import { createContext, useContext } from 'react';
const DarkModeContext = createContext(false); #1
...
function App() {
  const [isDarkMode, setDarkMode] = useState(false);
  ...
  return (
    <DarkModeContext.Provider value={isDarkMode}>
      ...
}
...
function SomeComponent() {
  const isDarkMode = useContext(DarkModeContext);
  ...
}
```

#1 Dark mode is disabled by default.

Nothing here should be too surprising. This example is how contexts work in JavaScript, and it works exactly the same way in a typesafe world. If we hover over the `DarkModeContext` variable in a TypeScript-capable editor, we can inspect the value and see that it is typed as `React.Context<boolean>`. The type of the context is inferred by the default value. TypeScript helps us provide the proper value. If we try to pass something non-Boolean to the provider, we'll get the usual TypeScript error about a type mismatch.

If we have a more complex context, such as an object with multiple values that we initialize to `null` but set it to a different value in the provider, we can provide a type argument to `createContext` about the expected type:

```
import { createContext, useContext } from 'react';
interface DarkMode {     #1
  mode: 'light' | 'dark';
  toggle: () => void;
}
const DarkModeContext =
  createContext<DarkMode | null>(null);    #2
function useDarkMode() {     #3
  const context = useContext(DarkModeContext);
  if (!context) {
    throw new Error(
      "useDarkMode must be wrapped in DarkModeContext.Provider"
    );
  }
  return context;
}
...
function App() {
  const [mode, setMode] = useState<"light" | "dark">("light");
  const toggle = () =>
    setMode((v) => (v === "light" ? "dark" : "light"));
  const contextValue = { mode, toggle };
  return (
    <DarkModeContext.Provider value={value}>     #4
      ...
}
...
function Button() {
  const { mode } = useDarkMode();   #5
  ...
}
```

#1 Defines the interface for the context contents

#2 Specifies the context type using a type argument

#3 Defines a custom hook that returns the correctly typed context. If the context value is null, the hook will throw; thus, it will never return null. The context value would only be null if we tried to access it outside the provider.

#4 Provides a value for the context that matches the interface for it

#5 Uses the context through the custom hook

Here, we're provide a type argument to `createContext` to specify the range of possible values we can pass in, making TypeScript aware that the value provided in the main app file has the expected format. You can see this typesafe variant of the dark mode application in the `dark-mode` example.

EXAMPLE: DARK-MODE

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

```
$ npm run dev -w ch06/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/ch06-dark-mode**.

In both examples, we never add type hints to `useContext`. This hook simply returns a value of the same type, as is contained in the context. So this hook is super simple to use in TypeScript land.

6.1.4 Typing effects

Effects are trivial to type, as an effect is a function that takes no arguments and returns a function or nothing. None of the effect hooks takes any type arguments because none is ever needed:

```
useEffect(() => {
  // Effect goes here - no types needed anywhere!
}, []);
```

You can use `useEffect` and `useLayoutEffect` the same way in TypeScript and JavaScript. The only type-related thing to be mindful of is the fact that the effect function is allowed to return only `undefined` or a function (the cleanup function). If you return any other value, TypeScript will tell you that you made an error.

Suppose that we have a timer component with an effect that runs an interval if the timer is running. You might write this code in regular JavaScript:

```
useEffect(() => {
  if (!isRunning) {
    return false;
  }
  const interval = setInterval(
    () => { /* do something */ },
    1000,
  );
  return () => clearInterval(interval);
}, []);
```

Using `return false;` this way out of habit, not because it does anything, is not unheard of, and it's fine in JavaScript. But in TypeScript, you'll receive this slightly cryptic error message:

```
Argument of type '() => false | (() => void)' is not assignable to
parameter of type 'EffectCallback'.
  Type 'false | (() => void)' is not assignable to type 'void |
  Destructor'.
    Type 'boolean' is not assignable to type 'void | Destructor'.
```

TypeScript complains that your function returns something other than `void` or a `Destructor`, which is an effect cleanup function. To fix the problem, simply change the returns to `return;` like so:

```
useEffect(() => {
  if (!isRunning) {
    return;
  }
  const interval = setInterval(
    () => { /* do something */ },
    1000,
  );
  return () => clearInterval(interval);
}, []);
```

This mistake isn't a common one, but you might come across the preceding weird error message. If so, the fix is to make sure to return only valid values (nothing or a cleanup function) from your effect.

6.1.5 Typing reducers

A well-behaved reducer is easy to type, as it’s a clean construction. The `useReducer` hook type follows the reducer in a straightforward manner. A small trick is required for generic reducers, however, because `useReducer` somehow forgets the generic argument.

Note In React 19, this situation has been fixed! No tricks are required, and you don’t need type hints for `useReducer` even when generics are involved.

Suppose that we’re setting out to build a revolutionary music-streaming application, bringing millions of songs to users worldwide. As part of our app’s features, we want to empower users to create their own music playlists, curating their favorite tracks into personalized lists. Crafting a seamless experience is paramount. Users should be able to rearrange the order of songs in their playlists easily, placing their most-loved tracks at the top or moving others to the bottom as they see fit.

To accomplish this task, we need a way to manage the state of these playlists in our application’s frontend. Instead of reinventing the wheel with every component or scenario, creating a custom hook dedicated to managing reorderable lists would be incredibly beneficial. This hook would encapsulate the logic needed to move songs up, down, or to specific positions within the playlist, ensuring a consistent and efficient user experience across the app. By building such a hook, which we’ll call `useReorderable`, we can take our application one step closer to delivering an unmatched music-curation experience to our users.

We could create this hook specifically to reorder song items in a playlist array, but why not make it generic so that it can reorder any type of element in a list of such elements? Let’s do just that. First, we define the state of the reducer, which is the list of elements:

```
type State<T> = T[];
```

Next, we need to define the actions as a discriminated union:

```
type Action =
  | { type: "moveUp"; index: number }
  | { type: "moveDown"; index: number }
  | { type: "moveToTop"; index: number }
  | { type: "moveToBottom"; index: number }
```

As you can see, the actions don't take a generic argument because the actions don't directly include the elements of the array—only manipulations done on those elements. Then, we need the main reducer function, which generally takes this form:

```
function reorder<T>(     #1
  state: State<T>,    #2
  action: Action   #3
): State<T> {     #4
  switch (action.type) {
    case "moveUp":
      ...
      // Return new array with the same elements in correct order
    ...
  }
}
```

#1 The reorder function is a reducer with a generic type argument, which can be any type.
#2 A reducer function always takes two arguments. The first argument is the current state of the data we're working on . . .
#3 . . . and the second argument is an action to apply to the current state.
#4 The reducer function returns the new state of the data, which may or may not be the same state as before.

At the end, we put the whole thing together in a custom hook, accepting the initial array and passing that and the reducer function into a `useReducer` call:

```
export function useReorderable<T>(initial: State<T>) {
  const [state, dispatch] = useReducer(reorder, initial);
  ...
}
```

Note that we don't add type hints to the `useReducer` function itself. We made the types perfectly clear when we created the reducer function, and as long as the type of the initial state matches the type of state the reducer works on, `useReducer` should be happy. Well, it is, but it isn't. If we hover over the state value in our editor, TypeScript reports the type as

```
const state: State<unknown>;
```

But that's wrong! The type is supposed to be `State<T>`, not `State<unknown>`. This result is where the trick comes in. We need to add a type hint to `useReducer` to inform it that the function is a reducer with a generic type argument.

Note In React 19, this situation has been improved. The state is correctly inferred as `State`<T> when you call `useReducer` without type arguments, so these shenanigans aren't required anymore.

React has a built-in interface, `Reducer`, that takes two type arguments: the state and the action interface, both of which are readily available. So instead of making the previous simple attempt to create our custom hook, we have to make this slightly more complicated version:

```
import { type Reducer, useReducer} from 'react';
...
export function useReorderable<T>(initial: State<T>) {
  const [state, dispatch] = useReducer<
    Reducer<State<T>, Action>
  >(reorder, initial);
```

When we put everything together, our custom hook, `useReorderable`, looks like the following listing.

Listing 6.2 The `useReorderable` hook

```
import { useReducer } from "react";
type State<T> = T[];
type Action =
  | { type: "moveUp"; index: number }
  | { type: "moveDown"; index: number }
  | { type: "moveToTop"; index: number }
  | { type: "moveToBottom"; index: number }
  | { type: "moveToPosition"; index: number; targetIndex: number };
function reorder<T>(state: State<T>, action: Action): State<T> {
  const newState: T[] = [...state];
  switch (action.type) {
    case "moveUp": {
      if (action.index <= 0 || action.index >= newState.length) {
        return state;
      }
      // Swap previous element with index
      [newState[action.index - 1], newState[action.index]] = [
        newState[action.index],
        newState[action.index - 1],
      ];
      return newState;
    }
    case "moveDown": {
      if (action.index < 0 || action.index >= newState.length - 1) {
        return state;
      }
      // Swap next element with index
      [newState[action.index], newState[action.index + 1]] = [
        newState[action.index + 1],
        newState[action.index],
      ];
      return newState;
    }
    case "moveToTop": {
      if (action.index <= 0 || action.index >= newState.length) {
        return state;
      }
      // Move index to top
      const itemToTop = newState.splice(action.index, 1)[0];
      newState.unshift(itemToTop);
```

```
        return newState;
      }
      case "moveToBottom": {
        if (action.index < 0 || action.index >= newState.length) {
          return state;
        }
        // Move index to bottom
        const itemToBottom = newState.splice(action.index, 1)[0];
        newState.push(itemToBottom);
        return newState;
      }
      default:
        return state;
    }
  }
  export function useReorderable<T>(initial: State<T>) {
    const [state, dispatch] = useReducer<
      Reducer<State<T>, Action>
    >(reorder, initial);
    return {
      list: state,
      moveUp: (index: number) =>     #1
        dispatch({ type: "moveUp", index }),   #1
      moveDown: (index: number) =>    #1
        dispatch({ type: "moveDown", index }),   #1
      moveToTop: (index: number) =>   #1
        dispatch({ type: "moveToTop", index }),  #1
      moveToBottom: (index: number) =>   #1
        dispatch({ type: "moveToBottom", index }),    #1
    };
  }
```

#1 We return a cleaner API than the raw dispatch function to make the hook easier to use.

Now let's get back to our revolutionary music-streaming application. We need to put this new hook to use. In the following listing, we have a playlist component that allows a user to reorder the songs in a playlist. We've added some nice icons and a CSS file for styling but didn't include the latter here.

Listing 6.3 A playlist component with items to reorder

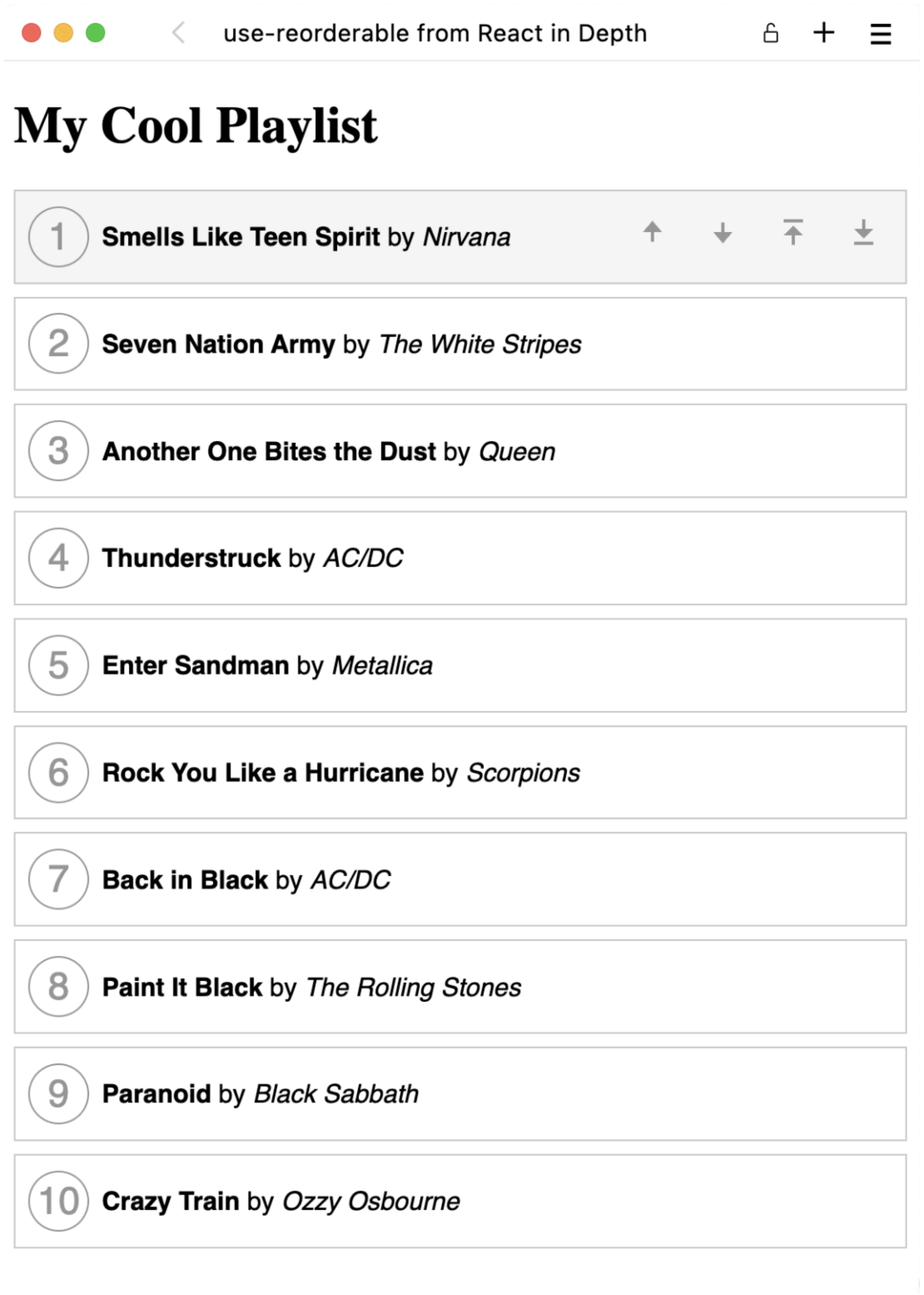
```
import {
  BiSolidUpArrowAlt as Up,
  BiSolidDownArrowAlt as Down,
  BiSolidArrowToTop as Top,
  BiSolidArrowToBottom as Bottom,
} from "react-icons/bi";
import { useReorderable } from "./useReorderable";
import "./playlist.css";
interface Song {
  id: number;
  title: string;
  artist: string;
}
interface PlaylistProps {
  songs: Song[];
}
export function Playlist({ songs }: PlaylistProps) {
  const {    #1
    list,    #1
    moveUp,    #1
    moveDown,    #1
    moveToBottom,    #1
    moveToTop,   #1
  } = useReorderable(songs);    #1

  return (
    <ol>
      {list.map((song, index) => (
        <li key={song.id}>
          <span>{index + 1}</span>
          <p>
            <strong>{song.title}</strong> by <em>{song.artist}</em>
          </p>
          <button onClick={() => moveUp(index)}>     #2
            <Up />    #2
          </button>    #2
          <button    #2
            onClick={() => moveDown(index)}    #2
          >    #2
            <Down />    #2
          </button>   #2
          <button    #2
            onClick={() => moveToTop(index)}    #2
          >    #2
            <Top />    #2
          </button>    #2
          <button    #2
            onClick={() => moveToBottom(index)} #2
          >    #2
            <Bottom />   #2
          </button>    #2
        </li>
      ))}
    </ol>
  );
}
```

#1 Uses the custom useReorderable hook exactly as defined

#2 Defines reorder buttons for each song. The CSS file will make them look nice.

In the main application, we pass in an array of actual songs. The result looks like figure 6.2.


use-reorderable from React in Depth

My Cool Playlist

1 **Smells Like Teen Spirit** by *Nirvana*

2 **Seven Nation Army** by *The White Stripes*

3 **Another One Bites the Dust** by *Queen*

4 **Thunderstruck** by *AC/DC*

5 **Enter Sandman** by *Metallica*

6 **Rock You Like a Hurricane** by *Scorpions*

7 **Back in Black** by *AC/DC*

8 **Paint It Black** by *The Rolling Stones*

9 **Paranoid** by *Black Sabbath*

10 **Crazy Train** by *Ozzy Osbourne*

Figure 6.2 My playlist after I prioritized songs a bit. All this fairly complex functionality is covered by a simple custom hook using a generic reducer and the properly typed `useReducer` hook. TypeScript is awesome.

EXAMPLE: USE-REORDERABLE

This example is in the `use-reorderable` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch06/use-reorderable
```

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/ch06-use-reorderable.

Now we can use this generic hook for reordering list items in other places. Do you want to have a list of your favorite album covers? Do you want to reorder the tracks currently queued up to play next? This example shows the power of custom hooks and TypeScript in action!

6.1.6 Typing memoization hooks

React has three memoization hooks, but only one of them—`useCallback`—is interesting from a TypeScript perspective. `useMemo` and `useDeferredValue` are identical in TypeScript compared with plain JavaScript.

TYPING USECALLBACK

Let's go back to the playlist, which is suboptimal from a performance perspective. Suppose that a user has 1,000 songs in a playlist and then swaps the positions of two of them. Because the rendering of every item happens directly in the playlist component, every item has to rerender. But for 998 of the songs, nothing changes, so we could have memoized them, with a great performance boost. To do this, we have to restructure things a bit by following these steps:

1. Create a new `PlaylistItem` component to handle a single song in a list.
2. Pass a single callback to the playlist item that takes both the index and the type of action to perform.

3. Rework the `useReorderable` hook so that it exports the dispatch function, as we're going to use it directly in the single callback.

Step 3 is the easiest one. We'll return the `useReducer` output directly (but still need the type hint from section 6.1.5):

```
export function useReorderable<T>(initial: State<T>) {
  return useReducer<Reducer<State<T>, Action>>(reorder, initial);
}
```

The following listing shows how we can implement the `PlaylistItem` component.

Listing 6.4 A single playlist-entry item

```
import {
  BiSolidUpArrowAlt as Up,
  BiSolidDownArrowAlt as Down,
  BiSolidArrowToTop as First,
  BiSolidArrowToBottom as Last,
} from "react-icons/bi";
import { Song } from "./types";    #1
import { MouseEvent, memo } from "react";
interface PlaylistItemProps {
  song: Song;
  index: number;
  move: (index: number) =>     #2
   (evt: MouseEvent<HTMLButtonElement>) => void;   #2
}
export const PlaylistItem =
memo(function PlaylistItem({    #3
  song,
  index,
  move,
}: PlaylistItemProps) {
  const onClick = move(index);
  return (
    <li>
      <span>{index + 1}</span>
      <p>
        <strong>{song.title}</strong> by <em>{song.artist}</em>
      </p>
      <button name="up" onClick={onClick}>    #4
        <Up />   #4
      </button>   #4
      <button name="down" onClick={onClick}>    #4
        <Down />   #4
      </button>   #4
      <button name="first" onClick={onClick}>   #4
        <First />   #4
      </button>  #4
      <button name="last" onClick={onClick}>   #4
        <Last />
      </button>
    </li>
    );
  });
```

#1 Notice that we've moved some commonly used types to a shared file.

#2 The single callback function has a slightly complicated signature, but it should make sense to you at this point.

#3 Defines the playlist item as a memoized component

#4 Uses the callback on each button identically. The action to perform will be deduced from the button name.

The only thing left to do is rewrite the main playlist component so that it uses the new `useReorderable` hook and then renders the list of items, passing in a callback for each item. First, let’s do this without memoization.

Listing 6.5 A simplified playlist component

```
import { useReorderable } from "./useReorderable";
import "./playlist.css";
import { ActionType, Song } from "./types";
import { PlaylistItem } from "./PlaylistItem";
interface PlaylistProps {
  songs: Song[];
}
export function Playlist({ songs }: PlaylistProps) {
  const [list, dispatch] = useReorderable(songs);
  return (
    <ol>
      {list.map((song, index) => (
        <PlaylistItem
          key={song.id}
          song={song}
          index={index}
          move={(index) => (evt) => {     #1
            const type =    #1
              evt.currentTarget.name as ActionType;   #1
            dispatch({ type, index });   #1
          }}    #1
        />
      ))}
    </ol>
  );
}
```

#1 Defines the move callback inline as it is passed to each playlist-entry item

When we define the callback inline, we don't need to add types to the arguments used in the functions because they can be inferred by the context in which the callback is used. TypeScript can infer the types from the type of the `move` property in the playlist item props interface.

But now we want to add memoization to the callback. Note that the callback is currently defined inside a loop, which we can't do with a hook, so we can't use `move={use Callback(...)}`. That attempt would violate the rules of hooks. So we have to define the function above the return statement, which also means that we have to add type information to the function arguments, as TypeScript no longer has the context of the `move` property near the definition. The following listing shows one solution.

Listing 6.6 The playlist with a memoized callback

```
import { useReorderable } from "./useReorderable";
import "./playlist.css";
import { ActionType, Song } from "./types";
import { PlaylistItem } from "./PlaylistItem";
import { MouseEvent, useCallback } from "react";
interface PlaylistProps {
  songs: Song[];
}
export function Playlist({ songs }: PlaylistProps) {
  const [list, dispatch] = useReorderable(songs);
  const handleMove = useCallback(
    (index: number) =>     #1
     (evt: MouseEvent<HTMLButtonElement>) => {   #1
      const type = evt.currentTarget.name as ActionType;
      dispatch({ type, index });
    },
    [dispatch]
  );
  return (
    <ol>
      {list.map((song, index) => (
        <PlaylistItem
          key={song.id}
          song={song}
          index={index}
          move={handleMove}
        />
      ))}
    </ol>
  );
}
```

#1 Adds type information to the function arguments inline inside the useCallback hook

EXAMPLE: PLAYLIST

This example is in the `playlist` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch06/playlist
```

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/ch06-playlist**.

The result is a fully optimized playlist-reordering application. If you try to check re-rendering by using React Developer Tools for the latest version (linked in the “Example: playlist” sidebar), you’ll see that if you swap two playlist entries, only those entries will re-render; the rest will be unchanged.

As always, there’s more than one way to skin a cat. All of the following statements are functionally identical ways to add type information to the callback:

```
// Define the function arguments in the function definition
const handleMove = useCallback(
  (index: number) => (evt: MouseEvent<HTMLButtonElement>) => {
    const type = evt.currentTarget.name as ActionType;
    dispatch({ type, index });
  },
  [dispatch]
);
// Define the function type generally as a type argument to useCallback
const handleMove = useCallback<
  (index: number) => (evt: MouseEvent<HTMLButtonElement>) => void
>(
  (index) => (evt) => {
    const type = evt.currentTarget.name as ActionType;
    dispatch({ type, index });
  },
  [dispatch]
);
// Defines the function type on the target variable
const handleMove: (
  index: number
) => (evt: MouseEvent<HTMLButtonElement>) => void = useCallback(
  (index) => (evt) => {
    const type = evt.currentTarget.name as ActionType;
    dispatch({ type, index });
  },
  [dispatch]
);
```

More abstractly, you can do the same thing in these three ways:

```
// Define the function arguments in the function definition
const callback = useCallback((a: Type) => {}, []);
// Define the function type generally as a type argument to useCallback
const callback = useCallback<(a: Type) => void>((a) => {}, []);
// Define the function type on the target variable
const callback:(a: Type) => void = useCallback((a) => {}, []);
```

I prefer the first method, but you might come across the other variants in the wild.

TYPING USEMEMO AND USEDEFERREDVALUE

`useMemo` and `useDeferredValue` are even easier to work with, as the value returned from each hook simply has the same type as the value returned by the callback passed to the hook. These callbacks never take arguments, which makes things even simpler. In short, you use these hooks the same way in TypeScript and regular JavaScript.

Note As mentioned in chapter 3, with the introduction of the new React Compiler, memoization is no longer required once React Compiler becomes stable and standard in all projects. However that might take some time and many projects will not use React Compiler for quite some time so this information is still important to know.

6.1.7 Typing the remaining hooks

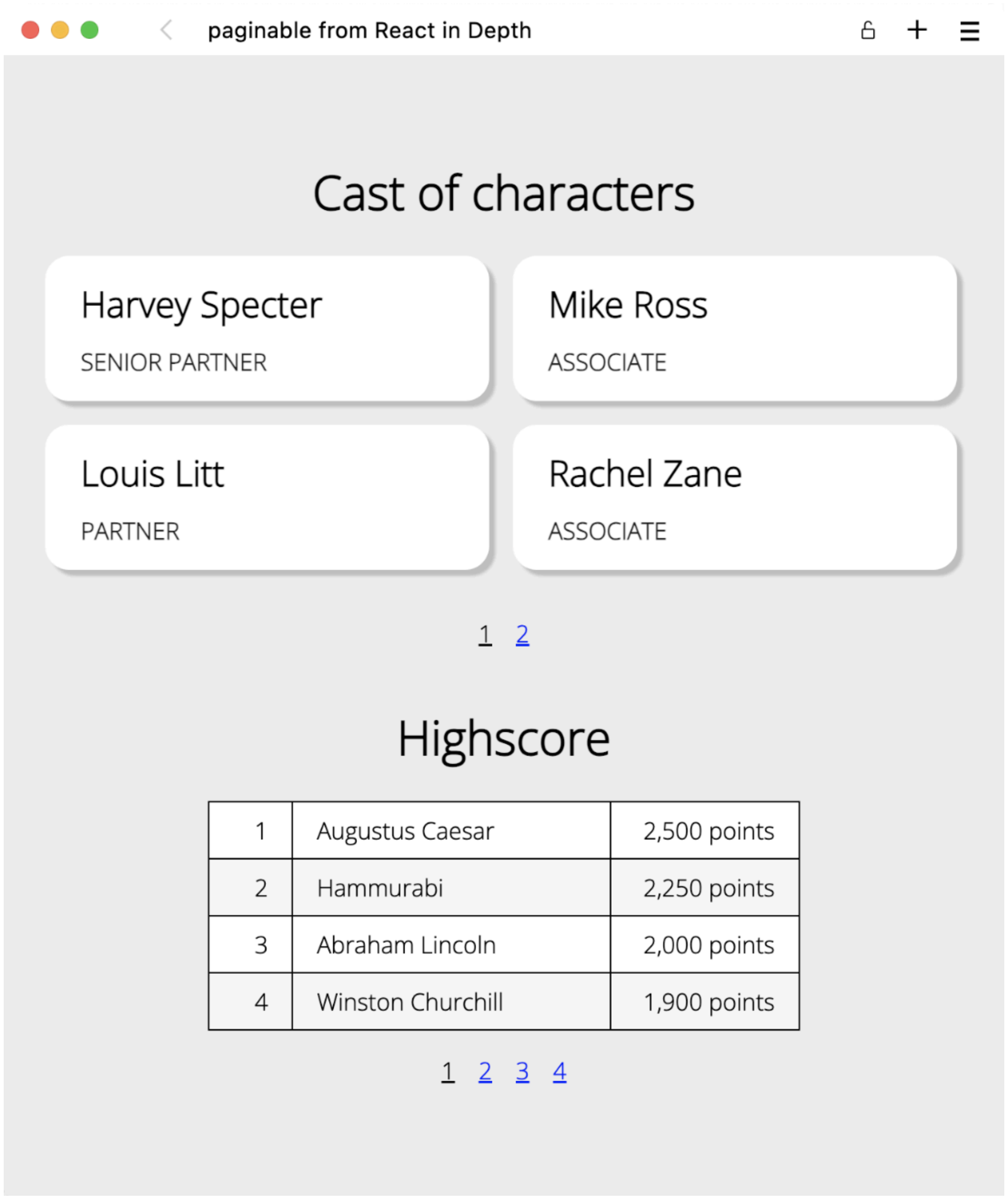
All the other hooks in React are seldom used and are extremely simple or quite complicated typewise, so we'll skip them for now. In the rare case in which you do need them in a TypeScript project, look them up in the React documentation, and you should be able to make sense of them.

6.2 Generic pagination: An example

So far, we've seen a ton of examples of using generics to make functions more versatile. Generics come up in hooks all the time. But components are functions, so can components be made generic too? Yes, they certainly can.

To create a useful React application by using generic components, we're going to create an application with a significant number of components, so it's not completely trivial. Grasping everything will require greater effort, but the result will be well worth the trouble.

We want to create a component that can display a list of items with pagination if there are many items. We want to display four items per page, and if there are more, we want little numbers below the component to allow the user to go to page 1 of the results, page 2 of the results, and so on (figure 6.3).



Cast of characters

Harvey Specter SENIOR PARTNER	Mike Ross ASSOCIATE
Louis Litt PARTNER	Rachel Zane ASSOCIATE

1 2

Highscore

1	Augustus Caesar	2,500 points
2	Hammurabi	2,250 points
3	Abraham Lincoln	2,000 points
4	Winston Churchill	1,900 points

1 2 3 4

Figure 6.3 Pagination used for a list of employee cards and a high score. This application shows how to use generics to apply similar functionality to different types of objects, as long as they follow the proper interfaces.

We want to create this component so that it works with any type of content, however. We want to make it generic. We will pass the component a list of items of something, and we will also supply a renderer for each something that needs to be displayed. That something is what we want our generic type to be. We can use this pagination component to display a list of employees, using the `EmployeeCard` component that we created in chapter 5 (section 5.2.3). We can also use the same com-

ponent to display a `paginable` high-score list, as shown in figure 6.3.
The following listing shows how.

Listing 6.7 A generic pagination component

```
import { FC, useState } from "react";
import "./paginable.css";
const PERPAGE = 4;
interface PaginableProps<Type> {    #1
  items: Type[];    #2
  Renderer: FC<PaginableItemProps<Type>>;     #3
  className?: string;
}
interface PaginableItemProps<Type> {    #4
  item: Type;     #5
  index: number;    #6
}   #6
export function Paginable<Type>({     #7
  items,    #7
  Renderer,    #7
  className,   #7
}: PaginableProps<Type>) {    #7
  const [currentPage, setCurrentPage] = useState(0);
  const startOffset = currentPage * PERPAGE;
  const endOffset = (currentPage + 1) * PERPAGE;
  const subset = items.slice(startOffset, endOffset);
  const numPages = Math.ceil(items.length / PERPAGE);
  const pages = Array.from(Array(numPages)).map((k, v) => v);
  return (
    <section className={className}>
      {subset.map((item, index) => (
        <Renderer     #8
          key={index}    #8
          item={item}    #8
          index={index + startOffset}    #8
        />   #8
      ))}
      {numPages > 1 && (
        <ol className="pagination">
          {pages.map((page) => (
            <li className="pagination__item" key={page}>
              {page === currentPage ? (
                page + 1
              ) : (
                <button
```

```
                    className="pagination__link"
                    onClick={() => setCurrentPage(page)}
                  >
                    {page + 1}
                  </button>
                )}
              </li>
            ))}
          </ol>
        )}
      </section>
    );
  }
```

#1 Now things really start to get complicated. First, the type of the Paginable component will be generic, as it will vary with the type of props received.
#2 The first property of the paginable is a list of items of a given type.
#3 The second property is a functional renderer component, which takes props that match the type of elements in the array in the first property.
#4 These other props that the renderer component takes are themselves generic.
#5 First, the renderer gets an item of the given type.
#6 Second, the renderer receives an index, which is the position of the rendered item in the array.
#7 We put this type magic together when we require the received props of the component to conform to the specified interface.
#8 Finally, we can use the renderer component directly by invoking it with an item and an index because we specified that it has to accept those properties.

We can use this pagination component for an employee list in an application like this:

```
import { EmployeeCard } from "./EmployeeCard";
import { Paginable } from "./Paginable";
import { Employee } from "./types";
const EMPLOYEES: Employee = [
  { name: "Harvey Specter", title: "Senior Partner" },
  ...,
];
function App() {
  return (
    <main>
      <h1>Cast of characters</h1>
      <Paginable
        className="employee-list"
        items={EMPLOYEES}
        Renderer={EmployeeCard}
      />
    </main>
  );
}
export default App;
```

Before we go see this application in action, let’s create something else to paginate so we can see generics at play. Let’s create a high-score list with entries, each with a position, name, and score. We create the entry as shown in the following listing.

Listing 6.8 A high-score entry list

```
import "./highscore.css";
import { Entry } from "./types";    #1
interface EntryProps {
  item: Entry;
  index: number;    #2
}
export function HighscoreEntry({ item, index }: EntryProps) {
  return (
    <div className="highscore-entry">
      <p className="highscore-entry__pos">
        {index + 1}    #3
      </p>
      <p className="highscore-entry__name">
        {item.name}    #4
      </p>
      <p className="highscore-entry__points">
        {item.points.toLocaleString("en-US")}     #5
        points
      </p>
    </div>
  );
}
```

#1 A high-score entry consists of a name and some point value, which is a type we import from a central file.
#2 For this component, we also need the index of the entry in the overall list, as that will be the position in the high-score list.
#3 The index is incremented by 1, as the top spot in a high-score list is generally called #1, not #0.
#4 The name is printed as is.
#5 The points are formatted with a bit of built-in JavaScript functionality to add thousands separators to the list.

The following listing shows two different paginable components.

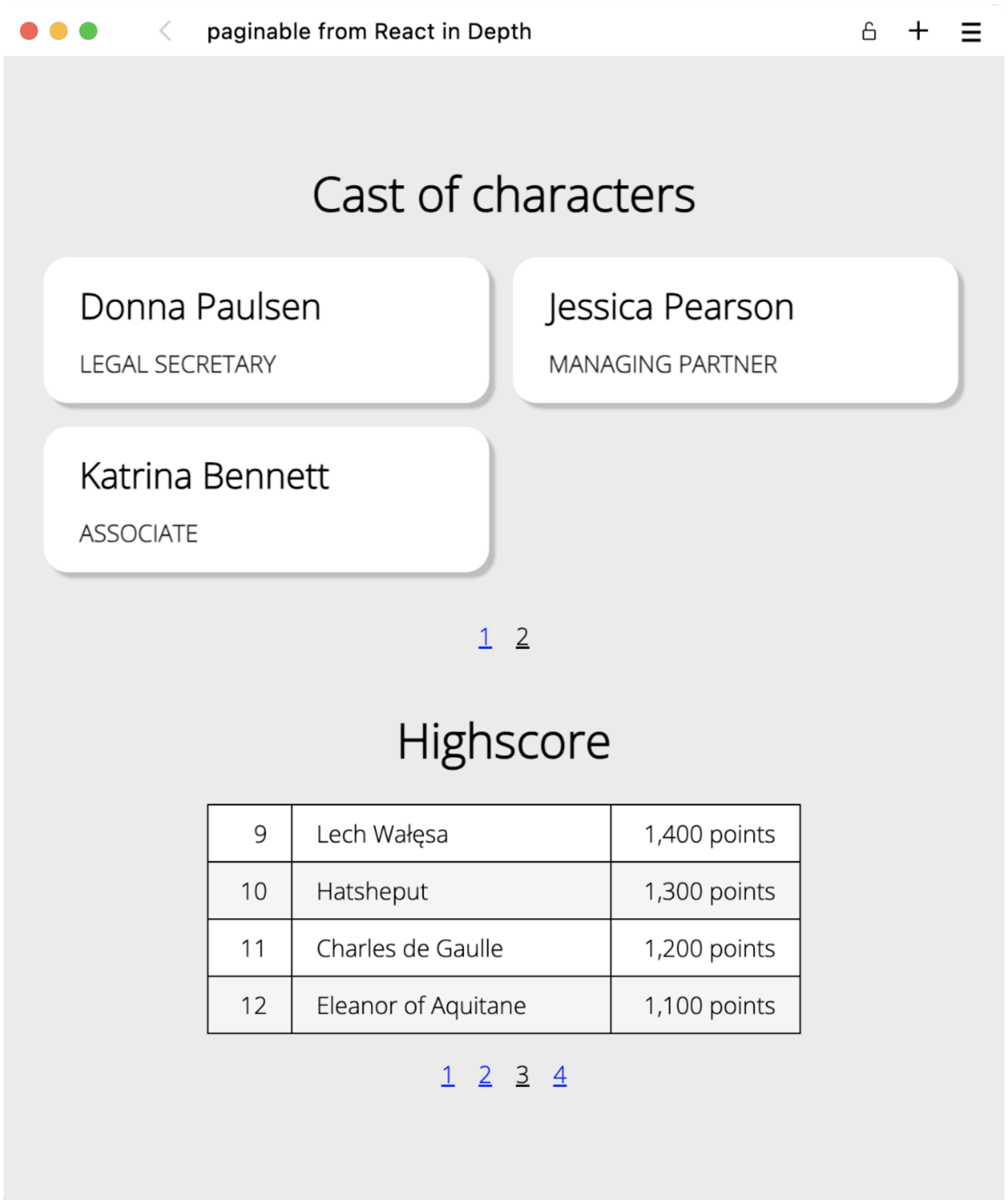
Listing 6.9 The main application

```
import "./app.css";
import { EmployeeCard } from "./EmployeeCard";
import { HighscoreEntry } from "./HighscoreEntry";
import { Paginable } from "./Paginable";
import { Employee, Entry } from "./types";
const EMPLOYEES: Employee[] = [
  { name: "Harvey Specter", title: "Senior Partner" },
  { name: "Mike Ross", title: "Associate" },
  { name: "Louis Litt", title: "Partner" },
  { name: "Rachel Zane", title: "Associate" },
  { name: "Donna Paulsen", title: "Legal Secretary" },
  { name: "Jessica Pearson", title: "Managing Partner" },
  { name: "Katrina Bennett", title: "Associate" },
];
const ENTRIES: Entry[] = [
  { name: "Augustus Caesar", points: 2500 },
  { name: "Hammurabi", points: 2250 },
  { name: "Abraham Lincoln", points: 2000 },
  { name: "Winston Churchill", points: 1900 },
  { name: "Nelson Mandela", points: 1800 },
  { name: "Catherine the Great", points: 1700 },
  { name: "Ashoka", points: 1600 },
  { name: "Marcus Aurelius", points: 1500 },
  { name: "Lech Wałęsa", points: 1400 },
  { name: "Hatsheput", points: 1300 },
  { name: "Charles de Gaulle", points: 1200 },
  { name: "Eleanor of Aquitane", points: 1100 },
  { name: "Ivan the Terrible", points: 1000 },
];
function App() {
  return (
    <main>
      <h1>Cast of characters</h1>
      <Paginable    #1
        className="employee-list"    #1
        items={EMPLOYEES}   #1
        Renderer={EmployeeCard}   #1
      />    #1
      <h1>Highscore</h1>
      <Paginable     #2
```

```
          className="highscores"    #2
          items={ENTRIES}   #2
          Renderer={HighscoreEntry}    #2
        />    #2
      </main>
    );
  }
  export default App;
```

#1 We instantiate the Paginable component for the employee list first with all the correct props.
#2 We do the same thing for the high score. Note that TypeScript would warn us directly inline if any entry in the high-score list was wrong or if the HighscoreEntry component did not accept the correct properties.

Putting everything together in the browser, we get what we set out to get in figure 6.4. We can even navigate the two lists separately, such as with the cast list on page 2 and the high score on page 3 (figure 6.4).



Cast of characters

Donna Paulsen
LEGAL SECRETARY

Jessica Pearson
MANAGING PARTNER

Katrina Bennett
ASSOCIATE

1 2

Highscore

9	Lech Wałęsa	1,400 points
10	Hatsheput	1,300 points
11	Charles de Gaulle	1,200 points
12	Eleanor of Aquitane	1,100 points

1 2 3 4

Figure 6.4 Here, we see pagination in action as we did in figure 6.3. This time, we've progressed the page differently in the two lists, demonstrating that pagination happens independently in each component.

I strongly urge you to check out this example on your own, as you'll fully appreciate how TypeScript makes it possible only when you start to play around with it. The errors you get when typing something wrong are mind-blowing, and the code-completion hints you get when you might not expect them are helpful.

Note that this code is complicated. It's okay if you don't fully understand it; a lot is going on here. This example is meant only to inspire you to seek more information. As I mentioned, I could have dedicated

this whole book to TypeScript, and it would have been just as long and comprehensive.
Also, the rabbit hole goes a lot deeper. TypeScript has some crazy options that enable you to program your types with a programming syntax inside the type system to make dynamic, extensible types.
Remember, TypeScript never runs in the browser in the final application; it runs only in your editor and in your compiler to make sure that it will run correctly. What you can do with TypeScript, and what TypeScript can do for you, is quite amazing.

Generic components, such as `Paginable`, have one small gotcha. Two built-in functions in the React core API don't play well with generic components, and you need to work a bit around the problem (or hope that it will be fixed in a future release, which is likely).

If you memoize or forward a reference to a generic component, your type argument for the component will be "forgotten." Unfortunately, we have to fix that ourselves to get back the generic functionality, which we want to do.

6.2.1 Forwarding a reference to a generic component

Suppose that we want to allow the main application to add a reference to the `paginable` list in the `Paginable` example. We want to allow the parent component to pass in a reference to a section element and then assign that reference to `<section>` inside the component:

```
import { forwardRef, Ref, FC, useState } from "react";
...
export const Paginable = forwardRef(function Paginable<Type>(
  { items, Renderer, className }: PaginableProps<Type>,
  ref: Ref<HTMLElement>,
) {
  ...
  return (
    <section ref={ref} className={className}>
```

This example should work for this component, right? No. Whether or not we pass in a reference, we get a TypeScript error in our main application when we pass a `Renderer` property to `Paginable`:

```
Type '({ item }: CardProps) => Element' is not assignable to type
'FC<PagenableItemProps<unknown>>'.
  Types of parameters '__0' and 'props' are incompatible.
    Type 'PagenableItemProps<unknown>' is not assignable to type
    'CardProps'.
      Types of property 'item' are incompatible.
        Type 'unknown' is not assignable to type 'Employee'.
```

Suddenly, the type argument is `unknown` when we expected it to be the type of item that we're rendering: `Employee`. But when we apply `forwardRef` to the component, the type argument is somehow "forgotten" and replaced by `unknown`. Now we can no longer achieve our goal of matching the items with the renderer.

The problem is that `forwardRef` does not return a component with the same type that you pass into it. We do not expect exactly the same type, but just because we want to pass references to the component doesn't mean that we want to change anything else.

Two solutions are available. The first is good for one-off (or maybe two-off) occasions, but if you have many generic components, the latter might be your go-to:

- Cast the component to the correct type after applying `forwardRef`.
- Augment the type of `forwardRef` in the schema for React globally in your application.

To cast the type, we need to think about how a component without `forwardRef` is different from one with `forwardRef`. The only difference is that the component with a reference takes all the same properties and returns the same value, but it also takes a `ref` property typed to the specific type of element. To type such a function, still with a generic type, we need the following complex expression:

```
function PaginableWithoutRef<Type>(    #1
  { items, Renderer, className }: PaginableProps<Type>,
  ref: Ref<HTMLElement>
) {
...
}
const Paginable =
  forwardRef(PaginableWithoutRef) as <Type>(     #2
    p: PaginableProps<Type>    #2
      & RefAttributes<HTMLElement>    #2
) => ReturnType<typeof PaginableWithoutRef>;  #2
```

#1 Defines the original component in a separate statement

#2 Creates the exported component by wrapping the original in forwardRef and then casting it to a similar component that also allows ref attributes

This code looks clunky, mostly because it's overly complex and comes out of nowhere. But despite the way the code looks, it works. If you have to type the code several times, it becomes a bit annoying, and you might forget it. Also, the code can be confusing to new developers, so you should add a comment to explain what's going on.

An alternative to adding this type cast in every generic component with a forwarded `ref` is to augment the schema for the React namespace. Yes, we can do that. We can redefine the types for external libraries, including those of React, because we "know better" than React. The React team made an error in its types, and we are allowed to fix it and are capable of fixing it.

To augment the schema, we create a TypeScript definition file (`*.d.ts`) somewhere in the source tree, such as `<root>/react-augmented.d.ts`. Note that we have to do this only once globally; then it should work for all components. In that file, we include the following:

```
import React from "react"
declare module "react" {
  function forwardRef<T, P = {}>(
    render: (props: P, ref: ForwardedRef<T>) => ReactElement | null
  ): (props: P & RefAttributes<T>) => ReactElement | null;
}
```

This code does the same thing as for the single instance, but abstractly, so it works for all instances.

Note As I've mentioned a few times, `forwardRef` is no longer required in React 19. Not having to jump through these hoops to type a generic component is another great benefit of this improvement in React 19.

6.2.2 Memoizing a generic component

Memoizing a generic component has the same problems and the same fixes. If we tried to memoize the `Paginable` example this way,

```
import { memo, FC, useState } from "react";
...
const Paginable = memo(function Paginable<Type>(
  { items, Renderer, className }: PaginableProps<Type>
) {
```

we'd run into the same error as with the `forwardRef` example. The fixes are the same: cast to the proper return type or augment React globally.

The first option is even easier for memoized components, as they don't change their signature. A memoized component is identical to the original typewise, so we can cast to the type of the original:

```
function PaginableWithoutMemo<Type>(
   ...
const Paginable = memo(PaginableWithoutMemo)
  as typeof PaginableWithoutMemo;
```

This example looks better than the reference example, but it would still be annoying to duplicate. The augmentation solution looks like this:

```
import React from "react"
declare module "react" {
  function memo<T, R>(Component: (props: T) => R): (props: T) => R;
}
```

Note This type fix isn't included in React 19. This is probably due to the work on the new experimental React Compiler, which when fully ready removes the need for memoization altogether. Read more about React Compiler here: https://react.dev/learn/react-compiler.

You can merge this example with the `forwardRef` augmentation in the same file, if you're using both in your codebase:

```
import React from "react"
declare module "react" {
  function forwardRef<T, P = {}>(
    render: (props: P, ref: ForwardedRef<T>) => ReactElement | null
  ): (props: P & RefAttributes<T>) => ReactElement | null;
  function memo<T, R>(Component: (props: T) => R): (props: T) => R;
}
```

6.3 Drawbacks of using TypeScript

Although TypeScript is great and beloved by many developers, starting to use it is not consequence free. The drawbacks of TypeScript are complexity and learning curve, which are two sides of the same coin:

- New developers, even strong React developers, without the required level of TypeScript experience will have a longer time getting familiar with the codebase and might even require dedicated training.
- The project inevitably ends up with more lines of code. Those extra lines are there to ensure that the lines you already had have less

errors, so they shouldn't be a problem.

- Finally, it takes more time to write code in TypeScript than in JavaScript because you write more code and have to think a bit more about how you write that code. But with the result being a more bug-free application, you should make that time up easily in less maintenance.

Overall, the advantages win out over time, but if you cannot bear the investment (in terms of training and extra time spent) up front, you may want to delay introducing TypeScript to your codebase until you can make such a commitment.

Some people might argue that the extra value TypeScript provides is irrelevant if you're a great developer. Daniel Heinemeier Hansson of Ruby on Rails fame made that argument in a controversial post about dropping TypeScript at **https://mng.bz/y8ep**. But in that post, Hansson missed teamwork. On a team, nobody (not even the best 10× Distinguished Engineer) knows the entire codebase, and TypeScript is currently the most efficient way to get hints about the things you don't know, so you're able to work with it quickly and confidently. To me, and to most of the rest of the JavaScript community, TypeScript is a godsend.

6.4 TypeScript resources

The important aspects of using TypeScript are understanding it fully and using its strengths to make your applications more resilient. The following books are great ways to learn more about TypeScript in general:

- *TypeScript Quickly*, by Yakov Fain and Anton Moiseev (**https://www.manning.com/books/typescript-quickly**)
- *TypeScript in 50 Lessons*, by Stefan Baumgartner of *Smashing Magazine* (**https://typescript-book.com**).

If you want to learn more about React and TypeScript in combination, check out these resources:

- React and TypeScript, a five-hour online course by Steve Kinney at Frontend Masters (**https://mng.bz/MZqQ**)
- React & Redux in TypeScript Complete Guide, an online reference guide by various contributors hosted at GitHub (**https://mng.bz/aEGj**)

Summary

- React hooks, particularly `useState` and `useRef`, require nuanced typing in TypeScript for optimal utility.
- Typing the `useState` hook in TypeScript properly ensures correct state inference and prevents type errors during state updates.
- When typing `useRef` in TypeScript, being explicit with types prevents problems with type inference and mutability. Although shorthand methods are available, they might result in immutable references, which can't be updated directly.
- Contexts in TypeScript can be typed based on their expected values, ensuring type safety. Simple contexts have straightforward typing, and complex contexts may require interfaces and type arguments. `useContext` itself remains straightforward to use with inferred types from the provided context.
- Effects in TypeScript are easy to use, similar to JavaScript. The main typing consideration is ensuring that effects return only `undefined` or a cleanup function; any other return value will cause a TypeScript error.
- Using TypeScript with React's `useReducer` allows for the creation of typesafe, versatile hooks for managing complex state transitions in applications.
- Using generic reducers in React's `useReducer` with TypeScript can be challenging and requires extra type arguments, but it provides a robust solution for managing abstract state structures.
- In TypeScript, the `useCallback` hook can be typed in various ways: directly defining the function arguments, providing a type argument to `useCallback`, or setting the type on the target variable.
- Both `useMemo` and `useDeferredValue` in TypeScript function similarly to their JavaScript counterparts, with the return type being

consistent with the callback's output.

- Just as TypeScript can be used to create generic functions, React components (which are essentially functions) can also be made generic. Generic components allow us to create versatile components that can adapt to different types of content as long as they adhere to the expected interfaces.
- Although TypeScript enhances the development experience, some challenges occur when combining TypeScript's generic components with certain React functionalities, such as `forwardRef` and `memo`. When you use these functions with generic components, TypeScript may "forget" the type, causing type mismatches. Solutions involve casting the component to the correct type post-function application or augmenting React's type definitions to handle generics correctly.
- TypeScript can pose a learning curve for new developers, demanding an initial time investment and potentially specialized training, but TypeScript's hints and guidance enhance code collaboration, outweighing its complexity with valuable advantages.