11
Reusable Components

In this chapter, we will build a checklist component and use various patterns to make it highly reusable but still strongly typed.

We will start by using TypeScript **generics** to strongly type the data passed to the component. Then, we will use the **props spreading** pattern to make the component API-flexible, and allow consumers of the component to custom render parts of the component using the **render props** pattern. After that, we will learn how to make custom hooks and use this to extract logic for checked items and how to make the state within a component controllable to change the component's behavior.

We'll cover the following topics:

- Creating the project
- Using generic props
- Using props spreading
- Using render props
- Adding checked functionality
- Creating custom hooks
- Allowing the internal state to be controlled

Technical requirements

We will use the following technologies in this chapter:

- **Node.js** and **npm**: You can install them here: `https://nodejs.org/en/download/`.
- **Visual Studio Code**: You can install it here: `https://code.visualstudio.com/`.

All the code snippets in this chapter can be found online at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter11`.

Creating the project

In this section, we will create the project for the app we will build and its folder structure. The folder structure will be straightforward because it contains a single page with the checklist component we will build.

We will develop the app using Visual Studio Code as in previous chapters, so open Visual Studio Code and carry out the following steps:

1. Create the project using Create React App. See *Chapter 3*, *Setting up React and TypeScript*, if you can't remember the steps for this.
2. We will style the app with Tailwind CSS, so install this into the project and configure it. See *Chapter 5*, *Approaches to Styling Frontends*, if you can't remember the steps for this.

That completes the project setup.

Using generic props

In this section, we'll take some time to understand how to create our own generic types and also learn about the `keyof` TypeScript feature, which is useful for generic types. We will use this knowledge to build the first iteration of the checklist component with a generic type for its props.

Understanding generics

We have used generics throughout this book. For example, the `useState` hook has an optional generic parameter for the type of state variable:

```
const [visible, setVisible] = useState<boolean>()
```

Generic parameters in a function allow that function to be reusable with different types and be strongly typed. The following function returns the first element in an array, or `null` if the array is empty. However, the function only works with a `string` array:

```
function first(array: Array<string>): string | null {
  return array.length === 0 ? null : array[0];
}
```

Generics allows us to make this function usable with any type of array.

Generic functions

Although we have used generic functions throughout this book, we haven't created our own generic function yet. Generic type parameters are defined in angled brackets before the function's parentheses:

```
function someFunc<T1, T2, ...>(...) {
 ...
}
```

The name of a generic type can be anything you like, but it should be meaningful so that it is easy to understand.

Here is a generic version of the function we saw earlier. Now, it can work with arrays containing any type of element:

```
function first<Item>(array: Array<Item>): Item | null {
  return array.length === 0 ? null : array[0];
}
```

The function has a single generic parameter called `Item`, which is used in the type of the `array` function parameter, as well as the function's return type.

Generic types

Custom types can be generic as well. For a `type` alias, its generic parameters are defined in angled brackets after the type name:

```
type TypeName<T1, T2, …> = {
 ...
}
```

For example, the props of a React component can be generic. An example of a generic props type is as follows:

```
type Props<Item> = {
  items: Item[];
  ...
};
```

The `Props` type has a single generic parameter called `Item`, which is used in the type of the `items` prop.

Generic React components

Generic props can be integrated into a generic function to produce a generic React component. Here's an example of a generic `List` component:

```
type Props<Item> = {
  items: Item[];
};
export function List<Item>({ items }: Props<Item>) {
  ...
}
```

The `items` prop in the `List` component can now have any type, making the component flexible and reusable.

Now that we understand how to create a component with generic props, we will create the first iteration of the checklist component.

Creating a basic list component

We will now start to create our reusable component. In this iteration, it will be a basic list containing some primary and secondary text obtained from an array of data.

Carry out the following steps:

1. Start by creating a folder for the component called `Checklist` in the `src` folder. Then, create a file called `Checklist.tsx` in this folder.
2. Open `Checklist.tsx` and add the following `Props` type:

   ```
   type Props<Data> = {
     data: Data[];
     id: keyof Data;
     primary: keyof Data;
     secondary: keyof Data;
   };
   ```

   Here is an explanation of each prop:

   - The `data` prop is the data that drives the items in the list
   - The `id` prop is the property's name in each data item that uniquely identifies the item
   - The `primary` prop is the property's name in each data item that contains the main text to render in each item
   - The `secondary` prop is the property's name in each data item that includes the supplementary text to render in each item

This is the first time we have encountered the `keyof` operator in a type annotation. It queries the type specified after it for the property names and constructs a union type from them, so the type for `id`, `primary`, and `secondary` will be a union type of all the property names for each data item.

3. Next, start to implement the component function as follows:

```
export function Checklist<Data>({
  data,
  id,
  primary,
  secondary,
}: Props<Data>) {
  return (
    <ul className="bg-gray-300 rounded p-10">
      {data.map((item) => {
      })}
    </ul>
  );
}
```

The component renders a gray, unordered list element with rounded corners. We also map over the data items where we will eventually render each item.

4. We will start by implementing the function inside `data.map`. The function checks whether the unique identifier (`idValue`) is a string or number, and if not, it won't render anything. The function also checks that the primary text property (`primaryText`) is a string, and again, if not, doesn't render anything:

```
{data.map((item) => {
  const idValue = item[id] as unknown;
  if (
    typeof idValue !== 'string' &&
    typeof idValue !== 'number'
  ) {
    return null;
  }
  const primaryText = item[primary] as unknown;
  if (typeof primaryText !== 'string') {
    return null;
```

```
    }
    const secondaryText = item[secondary] as unknown;
  }
```

5. Finish the implementation by rendering the list item as follows:

```
{data.map((item) => {
  ...
  return (
    <li
      key={idValue}
      className="bg-white p-6 shadow rounded mb-4
        last:mb-0"
    >
      <div className="text-xl text-gray-800 pb-1">
        {primaryText}
      </div>
      {typeof secondaryText === 'string' && (
        <div className="text-sm text-gray-500">
          {secondaryText}
        </div>
      )}
    </li>
  );
})}
```

The list items are rendered with a white background and rounded corners. The primary text is rendered as large, gray text with the secondary text rendered much smaller.

6. Create a new file in the `Checklist` folder called `index.ts` and export the `Checklist` component into it:

```
export * from './Checklist';
```

This file will simplify `import` statements for the `Checklist` component.

7. The final step before seeing the component in action is to add it to the component tree in the app. Open `App.tsx` and replace the content within it with the following:

```
import { Checklist } from './Checklist';

function App() {
```

```
    return (
      <div className="p-10">
        <Checklist
          data={[
            { id: 1, name: 'Lucy', role: 'Manager' },
            { id: 2, name: 'Bob', role: 'Developer' },
          ]}
          id="id"
          primary="name"
          secondary="role"
        />
      </div>
    );
  }

  export default App;
```

We reference the `Checklist` component and pass some data into it. Notice how type-safe the `id`, `primary`, and `secondary` attributes are – we are forced to enter a valid property name with the data items.

8. Run the app by entering `npm start` in the terminal. The checklist component appears as shown here:

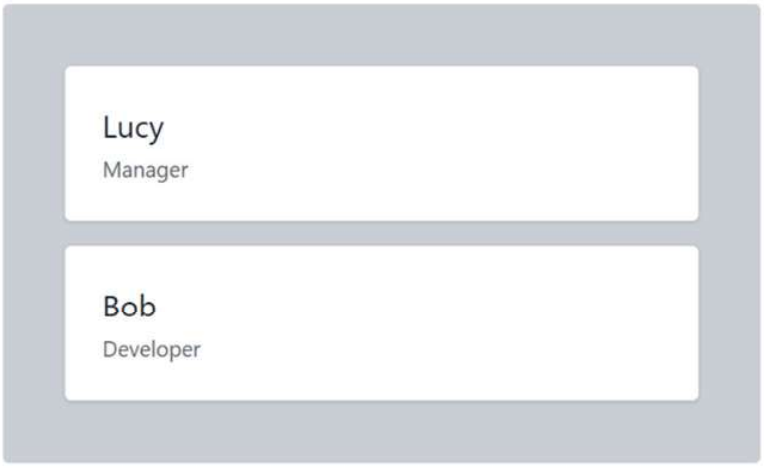


Figure 11.1 – Our basic checklist component

Currently, the component renders a basic list – we will add the checked functionality later in this chapter.

That completes this section on generic props.

To recap, here are some key points:

- TypeScript generics allow reusable code to be strongly typed.
- Functions can have generic parameters that are referenced within the implementation.
- Types can also have generic parameters that are referenced within the implementation.
- A React component can be made generic by feeding a generic props type into a generic function component. The component implementation will then be based on generic props.

Next, we will learn about a pattern that allows the prop type to inherit props from an HTML element.

Using props spreading

In this section, we'll learn about a pattern called **props spreading**. This pattern is useful when you want to use all the props from an HTML element in the implementation of a component. In our checklist component, we will use this pattern to add all the props for the `ul` element. This will allow consumers of the component to specify props, such as the height and width of the checklist.

So, carry out the following steps:

1. Open `Checklist.tsx` and import the following type from React:

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

   This type allows us to reference the props of an HTML element such as `ul`. It is a generic type that takes the HTML element name as a generic parameter.

2. Add the props from the `ul` element to the component props type as follows:

```
type Props<Data> = {
  data: Data[];
  id: keyof Data;
  primary: keyof Data;
  secondary: keyof Data;
} & ComponentPropsWithoutRef<'ul'>;
```

3. Add a **rest parameter** called `ulProps` to collect all the props for the `ul` element into a single `ulProps` variable:

```
export function Checklist<Data>({
  data,
  id,
```

```
    primary,
    secondary,
    ...ulProps
  }: Props<Data>) {
    ...
  }
```

This is the first time we have used rest parameters in this book. They collect multiple arguments that are passed into the function into an array, so any props that aren't called `data`, `id`, `primary`, or `secondary` will be collected into the `ulProps` array. For more information on rest parameters, see `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters`.

4. Now, we can spread `ulProps` onto the `ul` element using the spread syntax:

```
export function Checklist<Data>({
  data,
  id,
  primary,
  secondary,
  ...ulProps
}: Props<Data>) {
  return (
    <ul
      className="bg-gray-300 rounded p-10"
      {...ulProps}
    >...</ul>
  );
}
```

5. We can use this new feature of `Checklist` to specify the list height and width. Open `App.tsx` and add the following `style` attribute, as well as more data items:

```
<Checklist
  data={[
    { id: 1, name: 'Lucy', role: 'Manager' },
    { id: 2, name: 'Bob', role: 'Developer' },
    { id: 3, name: 'Bill', role: 'Developer' },
    { id: 4, name: 'Tara', role: 'Developer' },
    { id: 5, name: 'Sara', role: 'UX' },
```

```
      { id: 6, name: 'Derik', role: 'QA' }
    ]}
    id="id"
    primary="name"
    secondary="role"
    style={{
      width: '300px',
      maxHeight: '380px',
      overflowY: 'auto'
    }}
  />
```

6. If the app isn't running, run it by entering `npm start` in the terminal. The checklist component appears sized as we expect:

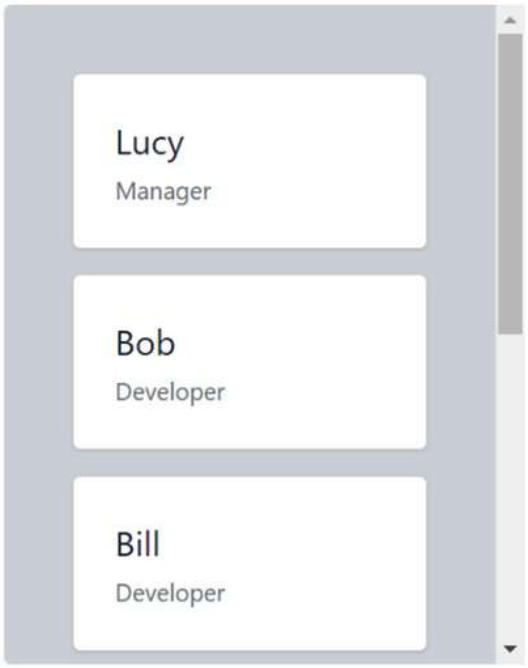


Figure 11.2 – The sized checklist component

The component now has a fixed height with a vertical scrollbar as a result of the style we passed into the component.

That completes our use of the props spreading pattern. Here's a recap of the key points:

- We intersect the props type with `ComponentPropsWithoutRef` to add props for the HTML element we want to spread onto
- We use a rest parameter in the component props to collect all the HTML element props into an array
- We can then spread the rest parameter onto the HTML element in the JSX

Next, we will learn about a pattern that allows consumers to render parts of a component.

Using render props

In this section, we will learn about the **render props** pattern and use it to allow the consumer of the component to render items within the checklist component.

Understanding the render props pattern

A way of making a component highly reusable is to allow the consumer to render internal elements within it. The `children` prop on a `button` element is an example of this because it allows us to specify any button content we like:

```
<button>We can specify any content here</button>
```

The render props pattern allows us to use a prop other than `children` to provide this capability. This is useful when the `children` prop is already used for something else, as in the following example:

```
<Modal heading={<h3>Enter Details</h3>}>
  Some content
</Modal>
```

Here, `heading` is a render prop in a `Modal` component.

Render props are useful when allowing the consumer to render elements associated with the data passed into the component because the render prop can be a function:

```
<List
  data={[...]}
  renderItem={(item) => <div>{item.text}</div>}
/>
```

The preceding example has a render prop called `renderItem` that renders each list item in a `List` component. The data item is passed into it so it can include its properties in the list item. This is similar to what we will implement next for our checklist component.

Adding a renderItem prop

We will add a prop called `renderItem` to the checklist that allows consumers to take control of the rendering of the list items. Carry out the following steps:

1. Open `Checklist.tsx` and add the `ReactNode` type to the React `import` statement:

```
import { ComponentPropsWithoutRef, ReactNode } from
'react';
```

`ReactNode` represents an element that React can render. Therefore, we will use `ReactNode` as the return type for our render prop.

2. Add a render prop called `renderItem` to the `Props` type:

```
type Props<Data> = {
  data: Data[];
  id: keyof Data;
  primary: keyof Data;
  secondary: keyof Data;
  renderItem?: (item: Data) => ReactNode;
} & React.ComponentPropsWithoutRef<'ul'>;
```

The prop is a function that takes in the data item and returns what needs rendering. We have made the prop optional because we will provide a default implementation for list items but also allow consumers to override it.

3. Add `renderItem` to the component function parameters:

```
export function Checklist<Data>({
  data,
  id,
  primary,
  secondary,
  renderItem,
  ...ulProps
}: Props<Data>) {
  ...
}
```

4. In the JSX, at the top of the mapping function, add an `if` statement to check whether the `renderItem` prop has been specified. If `renderItem` has been specified, call it with the data item, and return its result from the mapping function:

```
<ul ...>
  {data.map((item) => {
    if (renderItem) {
      return renderItem(item);
    }
    const idValue = item[id] as unknown;
    ...
  })}
</ul>
```

So, if `renderItem` has been specified, it will be called to get the element to render as the list item. If `renderItem` hasn't been specified, it will render the list item as it previously did.

5. To try the new prop out, open `App.tsx` and add the following `renderItem` attribute:

```
<Checklist
  ...
  renderItem={(item) => (
    <li key={item.id} className="bg-white p-4
      border-b-2">
      <div className="text-xl text-slate-800 pb-1">
        {item.name}
      </div>
      <div className="text-slate-500">{item.role}</div>
    </li>
  )}
/>
```

The list items are now rendered as flat, white items with a border between them.

6. If the app isn't running, run it by entering `npm start` in the terminal. The checklist component appears with the overridden list items:

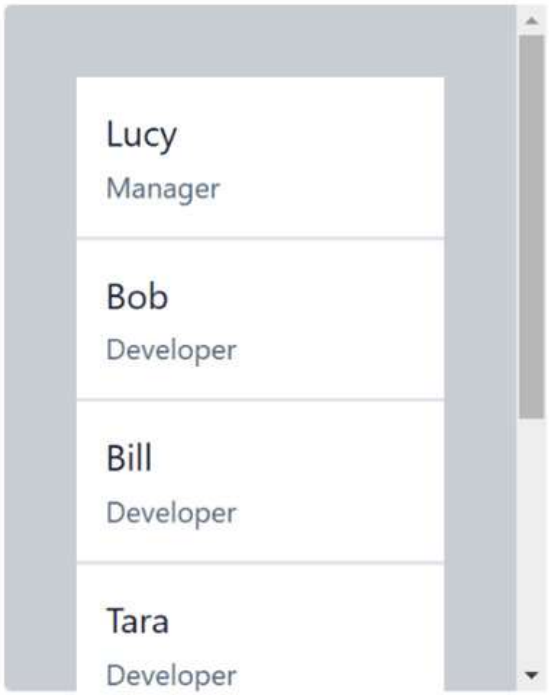


Figure 11.3 – Overridden list items

7. Before continuing to the next section, remove the use of `renderItem` in the `Checklist` element in `App.tsx`. The default rendering of list items should then appear.

That completes this section on the render props pattern. To recap, here are some key points:

- The render props pattern allows a component consumer to override the rendering of parts of the component
- A render prop can either be an element or a function that returns an element
- A common use case for a render prop is a data-driven list in which the rendering of list items can be overridden

Next, we will add checked functionality to our checklist component.

Adding checked functionality

Currently, our checklist component doesn't contain the ability to check items, so we will now add checkboxes to the list of items, giving users the ability to check them. We will track the checked items using a React state.

So, carry out the following steps to add this functionality to our component:

1. Open `Checklist.tsx` and add `useState` to the React `import` statement:

```
import {
  ComponentPropsWithoutRef,
  ReactNode,
  useState
} from 'react';
```

   We will use the state to store the IDs of the checked items.

2. At the top of the component implementation, add the state for the IDs of the checked items:

```
const [checkedIds, setCheckedIds] =
useState<IdValue[]>([]);
```

   We have referenced an `IdValue` type that we haven't defined yet – we'll define this after we have finished the component implementation in *step 6*.

3. Add checkboxes to the list of items as follows:

```
<li
  key={idValue}
  className="bg-white p-6 shadow rounded mb-4 last:mb-0"
>
  <label className="flex items-center">
    <input
      type="checkbox"
      checked={checkedIds.includes(idValue)}
      onChange={handleCheckChange(idValue)}
    />
    <div className="ml-2">
      <div className="text-xl text-gray-800 pb-1">
        {primaryText}
      </div>
      {typeof secondaryText === 'string' && (
```

```
          <div className="text-sm text-gray-500">
            {secondaryText}
          </div>
        )}
      </div>
    </label>
  </li>
```

The `checkedIds` state powers the `checked` attribute of the checkbox by checking whether the list item's ID is contained within it.

We will implement the referenced `handleCheckChange` function in the next step. Notice that the reference calls the function passing the ID of the list item that has been checked.

4. Start to implement the `handleCheckChange` function in the component as follows:

```
const [checkedIds, setCheckedIds] =
useState<IdValue[]>([]);
const handleCheckChange = (checkedId: IdValue) => () =>
{};
return ...
```

This is a function that returns the handler function. This complexity is because a basic checked handler doesn't pass in the list item's ID. This approach is called **currying**, and more information on it can be found at the following link: `https://javascript.info/currying-partials`.

5. Complete the handler implementation as follows:

```
const handleCheckChange = (checkedId: IdValue) => () => {
  const isChecked = checkedIds.includes(checkedId);
  let newCheckedIds = isChecked
    ? checkedIds.filter(
        (itemCheckedid) => itemCheckedid !== checkedId
      )
    : checkedIds.concat(checkedId);
  setCheckedIds(newCheckedIds);
};
```

The implementation updates the list item's ID to the `checkedIds` state if the list item has been checked and removes it if it is unchecked.

6. Next, let's define the `IdValue` type. Create a new file in the `Checklist` folder called `types.ts` with the definition of `IdValue`:

```
export type IdValue = string | number;
```

Here, the list item's ID can be a `string` or `number` value.

7. Move back to `Checklist.tsx` and import `IdValue`:

```
import { IdValue } from './types';
```

The compilation errors should now be resolved.

8. If the app isn't running, run it by entering `npm start` in the terminal. The checklist component appears with checkboxes for each list item:

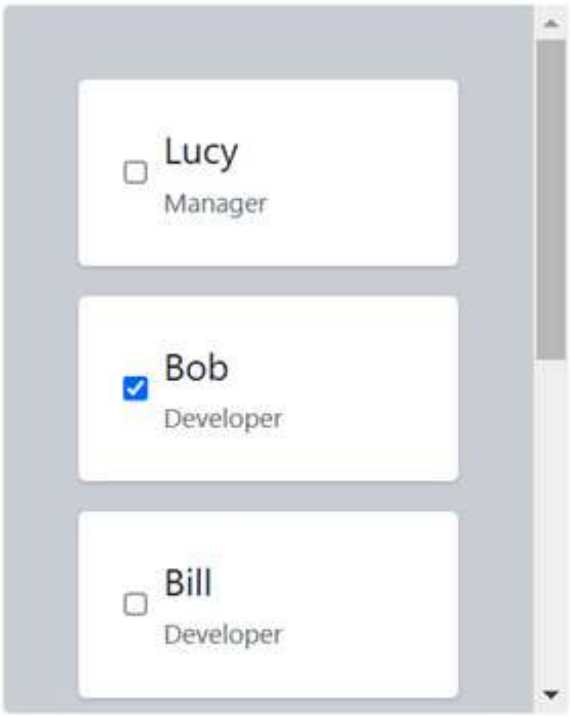


Figure 11.4 – Checkboxes for list items

The checklist component now includes checkboxes. However, there is an opportunity to make the checked logic reusable – we'll cover this in the next section.

Creating custom hooks

In this section, we'll learn about custom React hooks. Then, we will use this knowledge to extract the checked logic from the checklist component into a reusable custom hook.

Understanding custom hooks

As well as standard hooks such as `useState`, React allows us to create our own custom hooks. Custom hooks allow logic to be isolated and reused across multiple components.

A custom hook is defined using a function with a name that starts with the word *use*. This naming convention helps ESLint check for problems with the use of the custom hook. Here's a custom hook that provides toggling logic:

```
export function useToggle() {
  const [toggleValue, setToggleValue] = useState(false);

  function toggle() {
    setToggleValue(!toggleValue);
  }

  return {toggleValue, toggle};
};
```

The custom hook contains the state of the current toggle value, which is either `true` or `false`. It also includes a function called `toggle`, which toggles the current value. The current toggle value and the `toggle` function are returned from the custom hook in an object structure.

Note that an object structure doesn't have to be returned. If the custom hook only returns a single item, then that item can be returned directly. If the custom hook returns two things (as in the preceding example), it can return a tuple (as `useState` does). An object structure is better for more than two items because the object keys make it clear what each item is.

Another trait of a custom hook is that it uses other standard React hooks. For example, the `useToggle` custom hook uses `useState`. If the custom hook doesn't call a React hook or another custom hook, it's just a regular function rather than a custom hook.

This custom hook can then be used in the implementation of a component as follows:

```
const { toggleValue, toggle } = useToggle();
return (
  <div className="App">
    <button onClick={toggle}>{toggleValue ? 'ON' : 'OFF'}</
```

```
button>
  </div>
);
```

The toggle value (`toggleValue`) and the `toggle` function are destructured from the return value of the custom hook. The toggle value is used to render text **ON** or **OFF** inside the button content depending on whether it is `true` or `false`. The `toggle` function is also assigned to the click handler of the button.

Custom hooks can take in parameters as well. In the example here, we have added a default value in the `useToggle` hook:

```
type Params = {
  defaultToggleValue?: boolean;
};
export function useToggle({ defaultToggleValue }: Params) {
  const [toggleValue, setToggleValue] = useState(
    defaultToggleValue
  );
  ...
}
```

In the preceding example, the parameters are in an object structure. An object structure is nice when there are multiple parameters and nothing breaks when new parameters are added.

Arguments are passed into the custom hook in an object. Here's an example of using `useToggle` with its value initially being `true`:

```
const { toggleValue, toggle } = useToggle({
  defaultToggleValue: true
});
```

Now that we understand how to create and use custom hooks, we will put this into practice in our checklist component.

Extracting checked logic into a custom hook

We will extract the logic for checked items into a custom hook. This will allow potential future components to use the logic and clean up the code a little.

The custom hook will be called `useChecked` and will contain the state for the checked list item IDs. The hook will also include a handler that can be attached to the checkboxes, updating the checked list item ID's state.

To do this, carry out the following steps:

1. In the `Checklist` folder, create a file for the custom hook called `useChecked.ts`.
2. Open `useChecked.ts` and add the following `import` statements:

```
import { useState } from 'react';
import { IdValue } from './types';
```

The hook will use React state that is typed using `IdValue`.

3. Start to implement the function for the custom hook by initializing the state:

```
export function useChecked() {
  const [checkedIds, setCheckedIds] =
    useState<IdValue[]>([]);
}
```

The hook doesn't have any parameters. The `useState` call is exactly the same as the one currently in the `Checklist` component – this could be copied and pasted into the custom hook.

4. Add a checked handler to the custom hook. This can be copied from the implementation of the `Checklist` component:

```
export function useChecked() {
  const [checkedIds, setCheckedIds] =
    useState<IdValue[]>([]);

  const handleCheckChange = (checkedId: IdValue) => () =>
{
    const isChecked = checkedIds.includes(checkedId);
    let newCheckedIds = isChecked
      ? checkedIds.filter(
          (itemCheckedid) => itemCheckedid !== checkedId
        )
      : checkedIds.concat(checkedId);
    setCheckedIds(newCheckedIds);
  };
}
```

5. The last task in the custom hook implementation is to return the checked IDs and the handler function:

```
export function useChecked() {
  ...
  return { handleCheckChange, checkedIds };
}
```

6. Next, open `Checklist.tsx` and remove the state definition and the `handleCheckChange` handler function. Also, remove `useState` and `IdValue` from the `import` statements, as they are redundant.
7. Still in `Checklist.tsx`, import the `useChecked` hook we just created:

```
import { useChecked } from './useChecked';
```

8. Add a call to `useChecked` and destructure the checked IDs and the handler function:

```
export function Checklist<Data>({ ... }: Props<Data>) {
  const { checkedIds, handleCheckChange } = useChecked();
  return ...
}
```

9. If the app isn't running, run it by entering `npm start` in the terminal. The checklist component will appear and behave as it did before we made these changes.

That completes the implementation and use of the custom hook. To recap, here are some key points:

- Custom hooks make code a little cleaner and are reusable because they isolate logic, which can be complex.
- Custom hooks must start with `use`.
- Custom hooks must use a standard React hook or another custom hook.
- A custom hook is just a function that returns useful things for components to use. Using an object structure for the returned items is ideal when returning many items because the object keys make it clear what each item is.
- A custom hook can have parameters. Using an object structure for the parameters is ideal for many items and doesn't break anything when new parameters are added.

Next, we will cover a pattern that will allow the consumer of a component to control some of its behavior with the state.

Allowing the internal state to be controlled

In this section, we'll learn how to allow consumers of a component to control its internal state. We will use this pattern in the checklist component so that users can check just a single item.

Understanding how the internal state can be controlled

Allowing consumers of a component to control the state allows the behavior of a component to be tweaked if that behavior is driven by the state. Let's go through an example using the `useToggle` custom hook we covered in the last section when learning about custom hooks.

Two additional props are required to allow the internal state to be controlled – one for the current state value and one for a change handler. These additional props are `toggleValue` and `onToggleValueChange` in `useToggle`:

```
type Params = {
  defaultToggleValue?: boolean;
  toggleValue?: boolean;
  onToggleValueChange?: (toggleValue: boolean) => void;
};

export function useToggle({
  defaultToggleValue,
  toggleValue,
  onToggleValueChange,
}: Params) {
  ...
}
```

These props are marked as optional because this pattern doesn't force the consumer of the component to control the state – it's a feature they can opt in to.

> **Note**
>
> The consumer of the component will never specify both `defaultToggleValue` and `toggleValue`. `defaultToggleValue` should only be used when the consumer doesn't want to control `toggleValue` with the state. When the consumer does want to control `toggleValue` with the state, they can set the initial value of their state.

The `toggleValue` prop now clashes with the `toggleValue` state because they have the same name, so the state needs to be renamed:

```
const [resolvedToggleValue, setResolvedToggleValue] =
  useState(defaultToggleValue);

function toggle() {
  setResolvedToggleValue(!resolvedToggleValue);
}

return { resolvedToggleValue, toggle };
```

The default value of the internal state now needs to consider that there might be a prop controlling the state:

```
const [resolvedToggleValue, setResolvedToggleValue] =
  useState(defaultToggleValue || toggleValue);
```

When the state is changed, the change handler is called, if it has been defined:

```
function toggle() {
  if (onToggleValueChange) {
    onToggleValueChange(!resolvedToggleValue);
  } else {
    setResolvedToggleValue(!resolvedToggleValue);
  }
}
```

Again, it's important that we still update the internal state in case the consumer isn't controlling the state.

The last step in implementing this pattern is to update the internal state when the controlled state is updated. We can do this with `useEffect` as follows:

```
useEffect(() => {
  const isControlled = toggleValue !== undefined;
  if (isControlled) {
    setResolvedToggleValue(toggleValue);
  }
}, [toggleValue]);
```

The effect is triggered when the state prop changes. We check whether the state prop is being controlled; if so, the internal state is updated with its value.

Here's an example of controlling `toggleValue` in `useToggle`:

```
const [toggleValue, setToggleValue] = useState(false);
const onCount = useRef(0);
const { resolvedToggleValue, toggle } = useToggle({
  toggleValue,
  onToggleValueChange: (value) => {
    if (onCount.current >= 3) {
      setToggleValue(false);
    } else {
      setToggleValue(value);
      if (value) {
        onCount.current++;
      }
    }
  },
});
```

This example stores the toggle value in its own state and passes it to `useToggle`. `onToggleValueChange` is handled by updating the state value. The logic for setting the state value only allows it to be `true` up to three times.

So, this use case has overridden the default behavior of the toggle so that it can only be set to `true` up to three times.

Now that we understand how to allow the internal state to be controlled, we will use it in our checklist component.

Allowing checkedIds to be controlled

At the moment, our checklist component allows many items to be selected. If we allow the `checkedIds` state to be controlled by the consumer, they can change the checklist component so that they can select just a single item.

So, carry out the following steps:

1. We will start in `useChecked.ts`. Add `useEffect` to the React `import` statement:

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

2. Add new parameters for the controlled checked IDs and the change handler:

```
type Params = {
  checkedIds?: IdValue[];
  onCheckedIdsChange?: (checkedIds: IdValue[]) => void;
};
export function useChecked({
  checkedIds,
  onCheckedIdsChange,
}: Params) {
  ...
}
```

3. Update the internal state name to `resolvedCheckedIds` and default it to the passed-in `checkedIds` parameter if defined:

```
export function useChecked({
  checkedIds,
  onCheckedIdsChange,
}: Params) {
  const [resolvedCheckedIds, setResolvedCheckedIds] =
    useState<IdValue[]>(checkedIds || []);
  const handleCheckChange = (checkedId: IdValue) => () =>
{
    const isChecked = resolvedCheckedIds.
      includes(checkedId);
    let newCheckedIds = isChecked
      ? resolvedCheckedIds.filter(
          (itemCheckedid) => itemCheckedid !== checkedId
        )
      : resolvedCheckedIds.concat(checkedId);
    setResolvedCheckedIds(newCheckedIds);
  };
  return { handleCheckChange, resolvedCheckedIds };
}
```

4. Update the `handleCheckChange` handler to call the passed-in change handler if defined:

```
const handleCheckChange = (checkedId: IdValue) => () => {
  const isChecked = resolvedCheckedIds.
    includes(checkedId);
  let newCheckedIds = isChecked
    ? resolvedCheckedIds.filter(
        (itemCheckedid) => itemCheckedid !== checkedId
      )
    : resolvedCheckedIds.concat(checkedId);
  if (onCheckedIdsChange) {
    onCheckedIdsChange(newCheckedIds);
  } else {
    setResolvedCheckedIds(newCheckedIds);
  }
};
```

5. The last task in `useCheck.ts` is to synchronize the controlled checked IDs with the internal state. Add the following `useEffect` hook to achieve this:

```
useEffect(() => {
  const isControlled = checkedIds !== undefined;
  if (isControlled) {
    setResolvedCheckedIds(checkedIds);
  }
}, [checkedIds]);
```

6. Now, open `Checklist.tsx` and import the `IdValue` type:

```
import { IdValue } from './types';
```

7. Add the new props for the controlled checked IDs and the change handler:

```
type Props<Data> = {
  data: Data[];
  id: keyof Data;
  primary: keyof Data;
  secondary: keyof Data;
  renderItem?: (item: Data) => ReactNode;
  checkedIds?: IdValue[];
```

```
  onCheckedIdsChange?: (checkedIds: IdValue[]) => void;
} & ComponentPropsWithoutRef<'ul'>;

export function Checklist<Data>({
  data,
  id,
  primary,
  secondary,
  renderItem,
  checkedIds,
  onCheckedIdsChange,
  ...ulProps
}: Props<Data>) {}
```

8. Pass these props to `useChecked` and rename the destructured `checkedIds` variable `resolvedCheckedIds`:

```
const { resolvedCheckedIds, handleCheckChange } =
useChecked({
  checkedIds,
  onCheckedIdsChange,
});
return (
  <ul className="bg-gray-300 rounded p-10" {...ulProps}>
    {data.map((item) => {
      ...
      return (
        <li ... >
          <label className="flex items-center">
            <input
              type="checkbox"
              checked={resolvedCheckedIds.
                includes(idValue)}
              onChange={handleCheckChange(idValue)}
            />
            ...
          </label>
        </li>
```

```
        );
      })}
    </ul>
  );
```

9. Open `index.ts` in the `Checklist` folder. Export the `IdValue` type because consumers of the component can now pass in `checkedIds`, which is an array of this type:

```
export type { IdValue } from './types';
```

The `type` keyword after the `export` statement is required by TypeScript when exporting a named type already exported from the referenced file.

10. Now, open `App.tsx` and import `useState` from React, as well as the `IdValue` type:

```
import { useState } from 'react';
import {
  Checklist,
  IdValue
} from './Checklist';
```

11. Define the state in the `App` component for the single checked ID:

```
function App() {
  const [checkedId, setCheckedId] = useState<IdValue |
    null>(
    null
  );
  ...
}
```

The state is `null` when there is no checked item. This can't be set to `undefined` because `Checklist` will think `checkedIds` is uncontrolled.

12. Create a handler for when an item is checked:

```
function handleCheckedIdsChange(newCheckedIds: IdValue[])
{
  const newCheckedIdArr = newCheckedIds.filter(
    (id) => id !== checkedId
  );
  if (newCheckedIdArr.length === 1) {
    setCheckedId(newCheckedIdArr[0]);
```

```
    } else {
      setCheckedId(null);
    }
  }
```

The handler stores the checked ID in the state or sets the state to `null` if the checked item has been unchecked.

13. Pass the checked ID and the change handler to the `Checklist` element as follows:

```
<Checklist
  ...
  checkedIds={checkedId === null ? [] : [checkedId]}
  onCheckedIdsChange={handleCheckedIdsChange}
/>;
```

14. Let's give this a try. If the app isn't running, run it by entering `npm start` in the terminal. You will find that only a single list item can be checked.

That completes this section on allowing the internal state to be controlled. Here's a recap:

- This pattern is useful because it changes the component's behavior
- The component must expose a prop to control the state value and another for its change handler
- Internally, the component still manages the state and synchronizes it with the consumer's using `useEffect`
- If the state is controlled, the consumer's change handler is called in the internal change handler

Summary

In this chapter, we created a reusable checklist component and used many useful patterns along the way.

We started by learning how to implement generic props, which allow a component to be used with varying data types but still be strongly typed. We used this to allow varying data to be passed into the checklist component without sacrificing type safety.

We learned how to allow consumers of a component to spread props onto an internal element. A common use case is spreading props onto the internal container element to allow the consumer to size it, which is what we did with the checklist component.

The render prop pattern is one of the most useful patterns when developing reusable components. We learned that it allows the consumer to take responsibility for rendering parts of the component. We used this pattern to override the rendering of list items in our checklist component.

Custom hooks isolate logic and are useful for sharing logic across components and keeping the code within a component clean. Custom hooks must call a standard React hook directly or indirectly. We extracted the checked logic from our checklist component into a custom hook.

The last pattern we learned about was allowing a component's internal state to be controlled. This powerful pattern allows the consumer of the component to tweak its behavior. We used this to only allow a single list item to be checked in our checklist component.

In the next chapter, we will learn how to write automated tests for React components.

Questions

Answer the following questions to check what you have learned in this chapter:

1. The snippet of the following component renders options and one can be selected:

```
type Props<TOption> = {
  options: TOption[];
  value: string;
  label: string;
};
export function Select({
  options,
  value,
  label,
}: Props<TOption>) {
  return ...
}
```

   The following TypeScript error is raised on the component props parameter though: **Cannot find name 'TOption'**. What is the problem?

2. The `value` and `label` props from the component in *question 1* should only be set to a property name in the `options` value. What type can we give `value` and `label` so that TypeScript includes them in its type checking?

3. A prop called `option` has been added to the `Select` component from the previous question as follows:

```
type Props<TOption> = {
  ...,
  option: ReactNode;
```

```
};

export function Select<TOption>({
  ...,
  option
}: Props<TOption>) {
  return (
    <div>
      <input />
      {options.map((option) => {
        if (option) {
          return option;
        }
        return ...
      })}
    </div>
  );
}
```

`option` is supposed to allow the consumer of the component to override the rendering of the options. Can you spot the flaw in the implementation?

4. The following is a `Field` component that renders a `label` element and an `input` element:

```
type Props = {
  label: string;
} & ComponentPropsWithoutRef<'input'>;

export function Field({ ...inputProps, label }: Props) {
  return (
    <>
      <label>{label}</label>
      <input {...inputProps} />
    </>
  );
}
```

There is a problem with the implementation though – can you spot it?

5. How could the consumer specify props to spread onto the `label` element in the `Field` component from the previous question? Note that we still want the consumer to spread props onto the `input` element.
6. A custom hook has been added to the `Field` component from the previous question. The custom hook is called `useValid`, which validates that the field has been filled in with something:

```
export function useValid() {
  function validate(value: string) {
    return (
      value !== undefined && value !== null && value !==
        ''
    );
  }
  return validate;
}

export function Field({ ... }: Props) {
  const [valid, setValid] = useState(true);
  const validate = useValid();
  return (
    <>
      <label {...labelProps}>{label}</label>
      <input
        {...inputProps}
        onBlur={(e) => {
          setValid(validate(e.target.value));
        }}
      />
      {!valid && <span>Please enter something</span>}
    </>
  );
}
```

What is wrong with the implementation?

7. How many render props can a function component have?

Answers

1. The generic type must be defined in the component function as well as the prop:

```
export function Select<TOption>({
  options,
  value,
  label,
}: Props<TOption>) {
  return ...
}
```

2. The `keyof` operator can be used to ensure `value` and `label` are keys in `options`:

```
type Props<TOption> = {
  options: TOption[];
  value: keyof TOption;
  label: keyof TOption;
};
```

3. The consumer is likely to need the data for the option, so the prop should be a function containing the data as a parameter:

```
type Props<TOption> = {
  ...,
  renderOption: (option: TOption) => ReactNode;
};

export function Select<TOption>({
  options,
  value,
  label,
  renderOption,
}: Props<TOption>) {
  return (
    <div>
      <input />
      {options.map((option) => {
        if (renderOption) {
```

```
          return renderOption(option);
        }
        return ...
    </div>
  );
}
```

4. There is a syntax error because the rest parameter is the first parameter. The rest parameter must be the last one:

```
export function Field({ label, ...inputProps }: Props) {
  ...
}
```

5. A `labelProps` prop could be added using the `ComponentPropsWithoutRef` type. This could then be spread onto the `label` element:

```
type Props = {
  label: string;
  labelProps: ComponentPropsWithoutRef<'label'>;
} & ComponentPropsWithoutRef<'input'>;

export function Field({
  label,
  labelProps,
  ...inputProps
}: Props) {
  return (
    <>
      <label {...labelProps}>{label}</label>
      <input {...inputProps} />
    </>
  );
}
```

6. `useValid` doesn't call a standard React hook. A better implementation would be to extract the state into the custom hook as well:

```
export function useValid() {
  const [valid, setValid] = useState(true);
  function validate(value: string) {
    setValid(
      value !== undefined && value !== null && value !==
''
    );
  }
  return { valid, validate };
}

export function Field({ ... }: Props) {
  const { valid, validate } = useValid();
  return (
    <>
      <label {...labelProps}>{label}</label>
      <input
        {...inputProps}
        onBlur={(e) => {
          validate(e.target.value);
        }}
      />
      {!valid && <span>Please enter something</span>}
    </>
  );
}
```

7. There is no limit on the number of render props a component can have.

12

Unit Testing with Jest and React Testing Library

In this chapter, we learn how to use Jest and React Testing Library, two popular automated testing tools that can be used together in React applications. We will create tests on the checklist component we created in *Chapter 11*, *Reusable Components*.

We will start by focusing on Jest and using it to test simple functions, learning about Jest's common **matcher** functions for writing expectations, and how to execute tests to check whether they pass.

We will then move on to learning about component testing using React Testing Library. We'll understand the different query types and variants and how they help us create robust tests.

After that, we will learn the most accurate way to simulate user interactions using a React Testing Library companion package. We will use this to create tests for items being checked in the checklist component.

At the end of the chapter, we will learn how to determine which code is covered by tests and, more importantly, which code is uncovered. We use Jest's code coverage tool to do this and understand all the different coverage stats it gives us.

So, in this chapter, we'll cover the following topics:

- Testing pure functions
- Testing components
- Simulating user interactions
- Getting code coverage

Technical requirements

We will use the following technologies in this chapter:

- **Node.js** and **npm**: You can install them from `https://nodejs.org/en/download/`
- **Visual Studio Code**: You can install it from `https://code.visualstudio.com/`

We will start with a modified version of the code we finished in the last chapter. The modified code contains logic extracted into pure functions, which will be ideal to use in the first tests we write. This code can be found online at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter12/start`.

Carry out the following steps to download this to your local computer:

1. Go to `https://download-directory.github.io/` in a browser.
2. In the textbox on the web page, enter the following URL: `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter12/start`.
3. Press the *Enter* key. A ZIP file containing the `start` folder will now be downloaded.
4. Extract the ZIP file to a folder of your choice and open that folder in Visual Studio Code.
5. In Visual Studio Code's terminal, execute the following command to install all the dependencies:

```
npm i
```

You are now ready to start writing tests for the checklist component.

Testing pure functions

In this section, we will start by understanding the fundamental parts of a Jest test. Then, we will put this into practice by implementing tests on a pure function in the checklist component.

A pure function has a consistent output value for a given set of parameter values. These functions depend only on the function parameters and nothing outside the function, and also don't change any argument values passed into them. So, pure functions are nice for learning how to write tests because there are no tricky side effects to deal with.

In this section, we will also cover how to test exceptions, which is useful for testing type assertion functions. Finally, at the end of this section, we will learn how to run the tests in a test suite.

Understanding a Jest test

Jest is preinstalled in a Create React App project and configured to look for tests in files with particular extensions. These file extensions are `.test.ts` for tests on pure functions and `.test.tsx` for tests on components. Alternatively, a `.spec.*` file extension could be used.

A test is defined using Jest's `test` function:

```
test('your test name', () => {
  // your test implementation
});
```

The `test` function has two parameters for the test name and implementation. It is common practice for the test implementation to be an anonymous function. The test implementation can be asynchronous by placing the `async` keyword in front of the anonymous function:

```
test('your test name', async () => {
  // your test implementation
});
```

The test implementation will consist of calling the function with arguments being tested and checking the result is as we expect:

```
test('your test name', async () => {
  const someResult = yourFunction('someArgument');
  expect(someResult).toBe('something');
});
```

Jest's `expect` function is used to define our expectations. The result of the function call is passed into `expect`, and it returns an object containing methods we can use to define specific expectations for the result. These methods are referred to as **matchers**. If the expectation fails, Jest will fail the test.

The preceding test uses the `toBe` matcher. The `toBe` matcher checks that primitive values are equal, and the preceding test uses it to check that the `someResults` variable is equal to `"something"`. Other common matchers are as follows:

- `toStrictEqual` for checking the values in an object or array. This recursively checks every property in the object or array. Here's an example:

  ```
  expect(someResult).toStrictEqual({
    field1: 'something',
    field2: 'something else'
  });
  ```

- `not` for checking the opposite of a matcher. Here's an example:

  ```
  expect(someResult).not.toBe('something');
  ```

- `toMatch` for checking strings against **regular expressions** (**regexes**). Here's an example:

```
expect(someResult).toMatch(/error/);
```

- `toContain` for checking if an element is in an array. Here's an example:

```
expect(someResult).toContain(99);
```

A complete list of all the standard matchers can be found in the Jest documentation at `https://jestjs.io/docs/expect`.

Now that we understand the basics of a Jest test, we will create our first Jest test.

Testing isChecked

The first function we will test is `isChecked`. This function has two parameters:

- `checkedIds`: This is an array of IDs that are currently checked
- `idValue`: This is the ID to determine whether it is checked

We will write a test for when the list item is checked and another for when it isn't checked:

1. Create a file called `isChecked.test.ts` in the `src/Checklist` folder that will contain the tests.

> **Note**
> It is best practice to place test files adjacent to the source file being tested. This allows the developers to navigate to the test for a function quickly.

2. Open `isChecked.test.ts` and import the `isChecked` function:

```
import { isChecked } from './isChecked';
```

3. Start to create the first test as follows:

```
test('', () => {
});
```

Jest puts the `test` function in the global scope, so there is no need to import it.

4. Add the test name as follows:

```
test('should return true when in checkedIds', () => {
);
```

Forming a naming convention for test names is good practice so that they are consistent and easy to understand. Here, we have used the following naming structure:

should {expected output / behaviour} when {input / state condition}

5. Now, let's start to implement the logic inside the test. The first step in the test is to call the function being tested with the arguments we want to test:

```
test('should return true when in checkedIds', () => {
  const result = isChecked([1, 2, 3], 2);
});
```

6. The second (and last) step in the test is to check that the result is what we expect, which is `true` for this test:

```
test('should return true when in checkedIds', () => {
  const result = isChecked([1, 2, 3], 2);
  expect(result).toBe(true);
});
```

Since the result is a primitive value (a Boolean), we use the `toBe` matcher to verify the result.

7. Add a second test to cover the case when the ID isn't in the checked IDs:

```
test('should return false when not in checkedIds', () =>
{
  const result = isChecked([1, 2, 3], 4);
  expect(result).toBe(false);
});
```

That completes the tests on the `isChecked` function. Next, we will learn how to test exceptions that are raised. We will check that our tests work after that.

Testing exceptions

We are going to test the `assertValueCanBeRendered` type assertion function. This is a little different from the last function we tested because we want to test whether an exception is raised rather than the returned value.

Jest has a `toThrow` matcher that can be used to check whether an exception has been raised. For this to catch exceptions, the function being tested has to be executed inside the expectation, as follows:

```
test('some test', () => {
  expect(() => {
    someAssertionFunction(someValue);
```

```
  }).toThrow('some error message');
});
```

We will use this approach to add three tests on the `assertValueCanBeRendered` type assertion function. Carry out the following steps:

1. Create a file called `assertValueCanBeRendered.test.ts` in the `src/Checklist` folder for the tests and import the `assertValueCanBeRendered` type assertion function:

```
import { assertValueCanBeRendered } from './
assertValueCanBeRendered';
```

2. The first test we will add is to check whether an exception is raised when the value isn't a string or number:

```
test('should raise exception when not a string or
number', () => {
  expect(() => {
    assertValueCanBeRendered(
      true
    );
  }).toThrow(
    'value is not a string or a number'
  );
});
```

We pass the `true` Boolean value, which should cause an error.

3. Next, we will test whether an exception isn't raised when the value is a string:

```
test('should not raise exception when string', () => {
  expect(() => {
    assertValueCanBeRendered(
      'something'
    );
  }).not.toThrow();
});
```

We use the `not` matcher with `toThrow` to check that an exception is not raised.

4. The last test will test an exception isn't raised when the value is a number:

```
test('should not raise exception when number', () => {
  expect(() => {
```

```
      assertValueCanBeRendered(
        99
      );
    }).not.toThrow();
});
```

That completes the tests for the `assertValueCanBeRendered` type assertion function.

Now that we have implemented some tests, we will learn how to run them next.

Running tests

Create React App has an npm script called `test` that runs the tests. After the tests are run, a watcher will rerun the tests when the source code or test code changes.

Carry out the following steps to run all the tests and experiment with the test watcher options:

1. Open the terminal and execute the following command:

   ```
   npm run test
   ```

   `test` is a very common npm script, so the `run` keyword can be omitted. In addition, `test` can be shortened to `t`. So, a shortened version of the previous command is as follows:

   ```
   npm t
   ```

   The tests will be run, and the following summary will appear in the terminal:

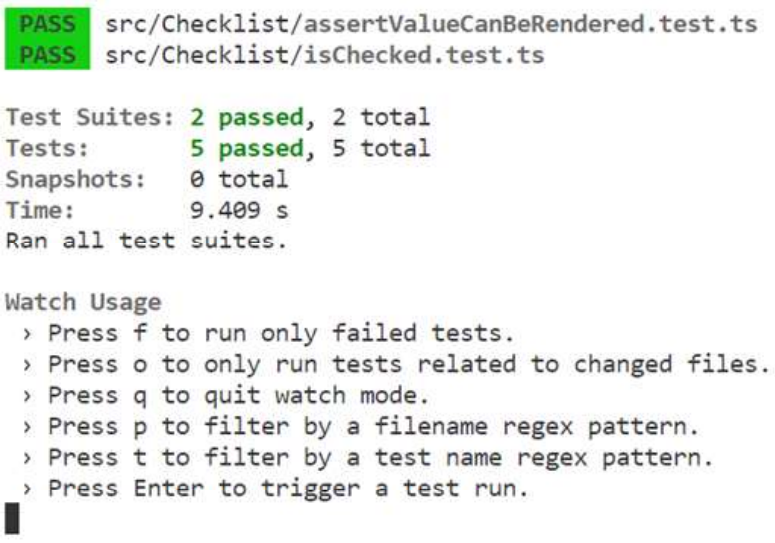


Figure 12.1 – First test run

Notice that there is no Command Prompt in the terminal like there usually is after a command has finished executing. This is because the command hasn't fully completed as the test watcher is running—this is called **watch mode**. The command won't complete until watch mode is exited using the *q* key. Leave the terminal in watch mode and carry on to the next step.

2. All the tests pass at the moment. Now, we will deliberately make a test fail so that we can see the information Jest provides us. So, open `assertValueCanBeRendered.ts` and change the expected error message on the first test as follows:

```
test('should raise exception when not a string or
number', () => {
  expect(() => {
    assertValueCanBeRendered(true);
  }).toThrow('value is not a string or a numberX');
});
```

As soon as the test file is saved, the tests are rerun, and a failing test is reported as follows:

```
FAIL src/Checklist/assertValueCanBeRendered.test.ts
  ● should raise exception when not a string or number

    expect(received).toThrow(expected)

    Expected substring: "value is not a string or a numberX"
    Received message:   "value is not a string or a number"

          5 | ): asserts value is IdValue {
          6 |   if (typeof value !== "string" && typeof value !== "number") {
        > 7 |     throw new Error("value is not a string or a number");
            |           ^
          8 |   }
          9 | }
         10 |
```

Figure 12.2 – Failing test

Jest provides valuable information about the failure that helps us quickly resolve test failures. It tells us this:

- Which test failed
- What the expected result was, in comparison to the actual result
- The line in our code where the failure occurred

Resolve the test failure by reverting the test to check for the correct error message. The test should be as follows now:

```
test('should raise exception when not a string or
number', () => {
  expect(() => {
    assertValueCanBeRendered(true);
  }).toThrow('value is not a string or a number');
});
```

3. We will now start to explore some of the options on the test watcher. Press the *w* key in the terminal, where the test watcher is still running. The test watcher options will be listed as follows:

```
Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press q to quit watch mode.
 › Press i to run failing tests interactively.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

Figure 12.3 – Test watcher options

4. We can filter the test files that Jest executes by using the *p* watch option. Press the *p* key and enter `isChecked` when prompted for the pattern. The pattern can be any regex. Jest will search for test files that match the regex pattern and execute them. So, Jest runs the tests in `isChecked.test.ts` in our test suite:

```
PASS src/Checklist/isChecked.test.ts
  √ should return true when in checkedIds (1 ms)
  √ should return false when not in checkedIds

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        6.234 s
Ran all test suites matching /isChecked/i.

Watch Usage: Press w to show more.
```

Figure 12.4 – Jest running a test file checking for a matching pattern

5. To clear the filename filter, press the *c* key.
6. We can also filter the tests that Jest executes by the test name using the *t* watch option. Press *t* and enter `should return false when not in checkedIds` when prompted for the test name. Jest will search for test names that match the regex pattern and execute them. So, Jest runs the `should return false when not in checkedIds` test in our test suite:

```
 PASS  src/Checklist/isChecked.test.ts

Test Suites: 1 skipped, 1 passed, 1 of 2 total
Tests:       4 skipped, 1 passed, 5 total
Snapshots:   0 total
Time:        8.05 s
Ran all test suites with tests matching "should return false when not in checkedIds".

Watch Usage: Press w to show more.
```

Figure 12.5 – Jest running test name matching a pattern

7. Press the *c* key to clear the test name filter and then press the *q* key to exit the test watcher.

That completes our exploration of running Jest tests and this section on testing pure functions. Here's a quick recap of the key points:

- Tests are defined using Jest's `test` function.
- Expectations within the test are defined using Jest's `expect` function in combination with one or more matchers.
- The `expect` function argument can be a function that executes the function being tested. This is useful for testing exceptions with the `toThrow` matcher.
- Jest's test runner has a comprehensive set of options for running tests. The test watcher is particularly useful on large code bases because it only runs tests impacted by changes by default.

Next, we will learn how to test React components.

Testing components

Testing components is important because this is what the user interacts with. Having automated tests on components gives us confidence that the app is working correctly and helps prevent regressions when we change code.

In this section, we will learn how to test components with Jest and React Testing Library. Then, we will create some tests on the checklist component we developed in the last chapter.

Understanding React Testing Library

React Testing Library is a popular companion library for testing React components. It provides functions to render components and then select internal elements. Those internal elements can then be checked using special matchers provided by another companion library called `jest-dom`.

A basic component test

Here's an example of a component test:

```
test('should render heading when content specified', () => {
  render(<Heading>Some heading</Heading>);
  const heading = screen.getByText('Some heading');
  expect(heading).toBeInTheDocument();
});
```

Let's explain the test:

- React Testing Library's `render` function renders the component we want to test. We pass in all the appropriate attributes and content so that the component is in the required state for the checks. In this test, we have specified some text in the content.
- The next line selects an internal element of the component. There are lots of methods on React Testing Library's `screen` object that allow the selection of elements. These methods are referred to as **queries**. `getByText` selects an element by matching the text content specified. In this test, an element with `Some heading` text content will be selected and assigned to the `heading` variable.
- The last line in the test is the expectation. The `toBeInTheDocument` matcher is a special matcher from `jest-dom` that checks whether the element in the expectation is in the DOM.

Understanding queries

A React Testing Library query is a method that selects a DOM element within the component being rendered. There are many different queries that find the element in different ways:

- `ByRole`: Queries elements by their role.

> **Note**
> DOM elements have a `role` attribute that allows assistive technologies such as screen readers to understand what they are. Many DOM elements have this attribute preset—for example, the `button` element automatically has the role of `'button'`. For more information on roles, see `https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles`.

- `ByLabelText`: Queries elements by their associated label. See this page in the React Testing Library documentation for the different ways elements can be associated with a label: `https://testing-library.com/docs/queries/bylabeltext`.
- `ByPlaceholderText`: Queries elements by their placeholder text.
- `ByText`: Queries elements by their text content.
- `ByDisplayValue`: Queries `input`, `textarea`, and `select` elements by their value.
- `ByAltText`: Queries `img` elements by their `alt` attribute.
- `ByTitle`: Queries elements by their `title` attribute.
- `ByTestId`: Queries elements by their test ID (the `data-testid` attribute).

There are also different types of queries that behave slightly differently on the found element. Each query type has a particular prefix on the query method name:

- `getBy`: Throws an error if a single element is not found. This is ideal for synchronously getting a single element.
- `getAllBy`: Throws an error if at least one element is not found. This is ideal for synchronously getting multiple elements.
- `findBy`: Throws an error if a single element is not found. The check for an element is repeated for a certain amount of time (1 second by default). So, this is ideal for asynchronously getting a single element that might not be immediately in the DOM.
- `findAllBy`: Throws an error if at least one element is not found within a certain time (1 second by default). This is ideal for asynchronously getting multiple elements that might not be immediately in the DOM.
- `queryBy`: This returns `null` if an element is not found. This is ideal for checking that an element does *not* exist.
- `queryAllBy`: This is the same as `queryBy`, but returns an array of elements. This is ideal for checking multiple elements do *not* exist.

So, the `getByText` query we used in the preceding test finds the element by the text content specified and raises an error if no elements are found.

For more information on queries, see the following page in the React Testing Library documentation: `https://testing-library.com/docs/queries/about/`.

Notice that none of these queries references implementation details such as an element name, ID, or CSS class. If those implementation details change due to code refactoring, the tests shouldn't break, which is precisely what we want.

Now that we understand React Testing Library, we will use it to write our first component test.

Implementing checklist component tests

The first component test we will write is to check that list items are rendered correctly. The second component test will check list items are rendered correctly when custom rendered.

React Testing Library and `jest-dom` are preinstalled in a Create React App project, which means we can get straight to writing the test. Carry out the following steps:

1. Create a new file in the `src/Checklist` folder called `Checklist.test.tsx` and add the following import statements:

```
import { render, screen } from '@testing-library/react';
import { Checklist } from './Checklist';
```

2. Start to create the test as follows:

```
test('should render correct list items when data
specified', () => {
});
```

3. In the test, render `Checklist` with some data:

```
test('should render correct list items when data
specified', () => {
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
    />
  );
});
```

We've rendered a single list item that should have primary text `Lucy` and secondary text `Manager`.

4. Let's check `Lucy` has been rendered:

```
test('should render correct list items when data
specified', () => {
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
```

```
        id="id"
        primary="name"
        secondary="role"
      />
    );
    expect(screen.getByText('Lucy')).toBeInTheDocument();
});
```

We have selected the element using the `getByText` query and fed that directly into the expectation. We use the `toBeInTheDocument` matcher to check that the found element is in the DOM.

5. Complete the test by adding a similar expectation for checking for `Manager`:

```
test('should render correct list items when data
specified', () => {
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
    />
  );
  expect(screen.getByText('Lucy')).toBeInTheDocument();
  expect(screen.getByText('Manager')).
    toBeInTheDocument();
});
```

That completes our first component test.

6. We will add the second test in one go, as follows:

```
test('should render correct list items when renderItem
specified', () => {
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
```

```
        renderItem={(item) => (
          <li key={item.id}>
            {item.name}-{item.role}
          </li>
        )}
      />
    );
    expect(
      screen.getByText('Lucy-Manager')
    ).toBeInTheDocument();
  });
```

We render a single list item with the same data as the previous test. However, this test custom renders the list items with a hyphen between the name and role. We use the same `getByText` query to check that the list item with the correct text is found in the DOM.

7. If the tests aren't automatically running, run them by running `npm test` in the terminal. Use the *p* option to run these two new tests—they should both pass:

```
PASS src/Checklist/Checklist.test.tsx
  √ should render correct list items when data specified (37 ms)
  √ should render correct list items when renderItem specified (5 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        6.593 s
Ran all test suites matching /Checklist\.test/i.

Watch Usage: Press w to show more.
```

Figure 12.6 – Component tests passing

That completes our first two component tests. See how easy React Testing Library makes this!

Using test IDs

The next test we will implement is to check that a list item is checked when specified. This test will be slightly trickier and requires a test ID on the checkboxes. Carry out the following steps:

1. Start by opening `Checklist.tsx` and notice the following test ID on the `input` element:

```
<input
  ...
```

```
    data-testid={`Checklist__input__${idValue.toString()}`}
  />
```

Test IDs are added to elements using a `data-testid` attribute. We can concatenate the list item ID so that the test ID is unique for each list item.

2. Now, return to the `Checklist.test.tsx` file and begin to write the test:

```
test('should render correct checked items when
specified', () => {
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
      checkedIds={[1]}
    />
  );
});
```

We have rendered the checklist with the same data as the previous tests. However, we have specified that the list item is checked using the `checkedIds` prop.

3. Now, on to the expectation for the test:

```
test('should render correct checked items when
specified', () => {
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
      checkedIds={[1]}
    />
  );
  expect(
    screen.getByTestId('Checklist__input__1')
  ).toBeChecked();
});
```

We select the checkbox by its test ID using the `getByTestId` query. We then use the `toBeChecked` matcher to verify the checkbox is checked. `toBeChecked` is another special matcher from the `jest-dom` package.

This new test should pass, leaving us with three passing tests on `Checklist`:

```
PASS src/Checklist/Checklist.test.tsx
  √ should render correct list items when data specified (37 ms)
  √ should render correct list items when renderItem specified (4 ms)
  √ should render correct checked items when specified (7 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        7.07 s
Ran all test suites matching /Checklist\.test/i.

Watch Usage: Press w to show more.
```

Figure 12.7 – All three component tests passing

4. Stop the test runner by pressing the *q* key.

That completes this section on testing components. Here's a quick recap:

- React Testing Library contains lots of useful queries for selecting DOM elements. Different query types will find single or many elements and will or won't error if an element isn't found. There is even a query type for repeatedly searching for elements rendered asynchronously.
- `jest-dom` contains lots of useful matchers for checking DOM elements. A common matcher is `toBeInTheDocument`, which verifies an element is in the DOM. However, `jest-dom` contains many other useful matchers, such as `toBeChecked` for checking whether an element is checked or not.

Next, we will learn how to simulate user interactions in tests.

Simulating user interactions

So far, our tests have simply rendered the checklist component with various props set. Users can interact with the checklist component by checking and unchecking items. In this section, we will first learn how to simulate user interactions in tests. We will then use this knowledge to test whether list items are checked when clicked and that `onCheckedIdsChange` is raised.

Understanding fireEvent and user-event

React Testing Library has a `fireEvent` function that can raise events on DOM elements. The following example raises a `click` event on a **Save** button:

```
render(<button>Save</button>);
fireEvent.click(screen.getByText('Save'));
```

This is okay, but what if logic was implemented using a `mousedown` event rather than `click`? The test would then need to be as follows:

```
render(<button>Save</button>);
fireEvent.mouseDown(screen.getByText('Save'));
```

Fortunately, there is an alternative approach to performing user interactions in tests. The alternative approach is to use the `user-event` package, which is a React Testing Library companion package that simulates user interactions rather than specific events. The same test using `user-event` looks like this:

```
const user = userEvent.setup();
render(<button>Save</button>);
await user.click(screen.getByText('Save'));
```

The test would cover logic implemented using a `click` event or `mousedown` event. So, it is less coupled to implementation details, which is good. For this reason, we'll use the `user-event` package to write interactive tests on our checklist component.

The `user-event` package can simulate interactions other than clicks. See the documentation at the following link for more information: `https://testing-library.com/docs/user-event/intro`.

Implementing checklist tests for checking items

We will now write two interactive tests on the checklist component. The first test will check items are checked when clicked. The second test will check `onCheckedIdsChange` is called when items are clicked. Carry out the following steps:

1. Create React App does preinstall the `user-event` package, but it may be a version before version 14, which has a different API. Open `package.json`, and then find the `@testing-library/user-event` dependency and check the version. If the version isn't 14 or above, then run the following command in the terminal to update it:

   ```
   npm i @testing-library/user-event@latest
   ```

2. We will add the interactive tests in the same test file as the other component tests. So, open `Checklist.test.tsx` and add an import statement for `user-event`:

```
import userEvent from '@testing-library/user-event';
```

3. The first test will test that items are checked when clicked. Start to implement this as follows:

```
test('should check items when clicked', async () => {
});
```

We have marked the test as asynchronous because the simulated user interactions in `user-event` are asynchronous.

4. Next, initialize the user simulation as follows:

```
test('should check items when clicked', async () => {
  const user = userEvent.setup();
});
```

5. We can now render a list item as we have done in previous tests. We will also get a reference to the checkbox in the rendered list item and check that it isn't checked:

```
test('should check items when clicked', async () => {
  const user = userEvent.setup();
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
    />
  );
  const lucyCheckbox = screen.getByTestId(
    'Checklist__input__1'
  );
  expect(lucyCheckbox).not.toBeChecked();
});
```

6. Now, on to the user interaction. Simulate the user clicking the list item by calling the `click` method on the `user` object; the checkbox to be clicked needs to be passed into the `click` argument:

```
test('should check items when clicked', async () => {
  const user = userEvent.setup();
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
    />
  );

  const lucyCheckbox = screen.getByTestId(
    'Checklist__input__1'
  );
  expect(lucyCheckbox).not.toBeChecked();
  await user.click(lucyCheckbox);
});
```

7. The last step in the test is to check that the checkbox is now checked:

```
test('should check items when clicked', async () => {
  const user = userEvent.setup();
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
    />
  );
  const lucyCheckbox = screen.getByTestId(
    'Checklist__input__1'
  );
```

```
    expect(lucyCheckbox).not.toBeChecked();
    await user.click(lucyCheckbox);
    expect(lucyCheckbox).toBeChecked();
  });
```

8. The next test will test that the function assigned to the `onCheckedIdsChange` prop is called when a list item is clicked. Here is the test:

```
test('should call onCheckedIdsChange when clicked', async
() => {
  const user = userEvent.setup();
  let calledWith: IdValue[] | undefined = undefined;
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
      onCheckedIdsChange={(checkedIds) =>
        (calledWith = checkedIds)
      }
    />
  );
  await user.click(screen.getByTestId('Checklist__
input__1'));
  expect(calledWith).toStrictEqual([1]);
});
```

We set a `calledWith` variable to the value of the `onCheckedIdsChange` parameter. After the list item is clicked, we check the value of the `calledWith` variable using the `toStrictEqual` matcher. The `toStrictEqual` matcher is a standard Jest matcher that is ideal for checking arrays and objects.

9. The second test references the `IdValue` type, so add an import statement for this:

```
import { IdValue } from './types';
```

10. Run the tests by running `npm test` in the terminal. Press the *p* key to run all the tests in the `Checklist.test.tsx` file. We should now have five passing component tests:

```
PASS  src/Checklist/Checklist.test.tsx
  √ should render correct list items when data specified (61 ms)
  √ should render correct list items when renderItem specified (10 ms)
  √ should render correct checked items when specified (11 ms)
  √ should check items when clicked (125 ms)
  √ should call onCheckedIdsChange when clicked (63 ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        7.958 s
Ran all test suites matching /Checklist\.test/i.

Watch Usage: Press w to show more.
```

Figure 12.8 – Five passing component tests

11. Stop the test runner by pressing the *q* key.

That completes the tests for clicking items and this section on simulating user interactions. We learned that React Testing Library's `fireAction` function raises a particular event that couples tests to implementation details. A better approach is to use the `user-event` package to simulate user interactions, potentially raising several events in the process.

Next, we will learn how to quickly determine any code that isn't covered by tests.

Getting code coverage

Code coverage is how we refer to how much of our app code is covered by unit tests. As we write our unit tests, we'll have a fair idea of what code is covered and not covered, but as the app grows and time passes, we'll lose track of this.

In this section, we'll learn how to use Jest's code coverage option so that we don't have to keep what is covered in our heads. We will use the code coverage option to determine the code coverage on the checklist component and understand all the different statistics in the report. We will use the code coverage report to find some uncovered code in our checklist component. We will then extend the tests on the checklist component to achieve full code coverage.

Running code coverage

To get code coverage, we run the `test` command with a `--coverage` option. We also include a `--watchAll=false` option that tells Jest not to run in watch mode. So, run the following command in a terminal to determine code coverage on our app:

```
npm run test -- --coverage --watchAll=false
```

The tests take a little longer to run because of the code coverage calculations. When the tests have finished, a code coverage report is output in the terminal with the test results:

```
------------------------------|---------|----------|---------|---------|-------------------
File                          | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
------------------------------|---------|----------|---------|---------|-------------------
All files                     |   57.44 |     62.5 |      60 |   57.77 |
 src                          |       0 |        0 |       0 |       0 |
  App.tsx                     |       0 |        0 |       0 |       0 | 5-14
  index.tsx                   |       0 |      100 |     100 |       0 | 7-19
  reportWebVitals.ts          |       0 |        0 |       0 |       0 | 3-10
 src/Checklist                |    93.1 |    93.75 |      90 |   92.85 |
  Checklist.tsx               |     100 |      100 |     100 |     100 |
  assertValueCanBeRendered.ts |     100 |      100 |     100 |     100 |
  getNewCheckedIds.ts         |      50 |       50 |      50 |      50 | 9-10
  index.ts                    |       0 |        0 |       0 |       0 |
  isChecked.ts                |     100 |      100 |     100 |     100 |
  types.ts                    |       0 |        0 |       0 |       0 |
  useChecked.ts               |     100 |      100 |     100 |     100 |
------------------------------|---------|----------|---------|---------|-------------------
```

Figure 12.9 – Terminal code coverage report

Next, we will take some time to understand this code coverage report.

Understanding the code coverage report

The coverage report lists the coverage for each file and aggregates coverage in a folder for all the files in the project. So, the whole app has between 57.44% and 62.5% code coverage, depending on which statistic we take.

Here's an explanation of all the statistic columns:

- **% Stmts**: This is **statement coverage**, which is how many source code statements have been executed during test execution
- **% Branch**: This is **branch coverage**, which is how many of the branches of conditional logic have been executed during test execution
- **% Funcs**: This is **function coverage**, which is how many functions have been called during test execution
- **% Lines**: This is **line coverage**, which is how many lines of source code have been executed during test execution

The rightmost column in the report is very useful. It gives the lines of source code that aren't covered by tests. For example, the `getNewCheckedIds.ts` file in the checklist component has lines 9 and 10, which are uncovered.

There is another version of the report that is generated in HTML format. This file is automatically generated every time a test is run with the `--coverage` option. So, this report has already been generated because we have just run the tests with the `--coverage` option. Carry out the following steps to explore the HTML report:

1. The report can be found in an `index.html` file in the `coverage\lcov-report` folder. Double-click on the file so that it opens in a browser:

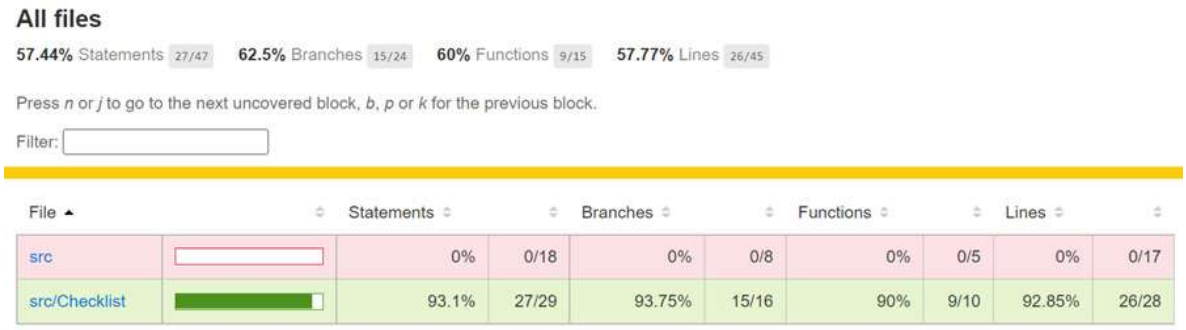


Figure 12.10 – HTML coverage report

   The report contains the same data as the terminal report, but this one is interactive.

2. Click on the **src/Checklist** link in the second row of the report. The page now shows the coverage for the files in the checklist component:

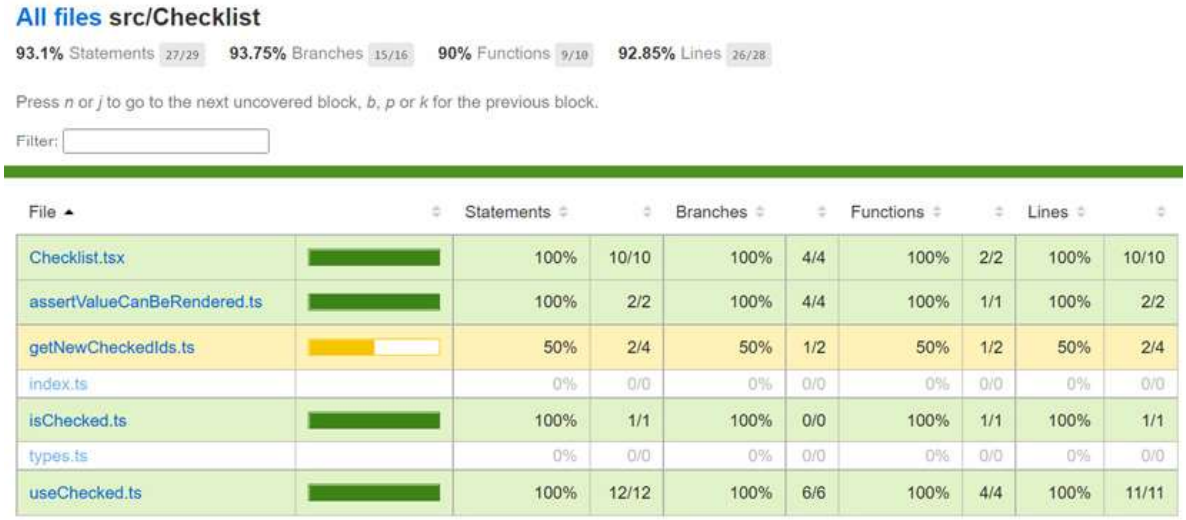


Figure 12.11 – Coverage report for checklist component files

3. Click on the `getNewCheckedIds.ts` link to drill into the coverage for that file:

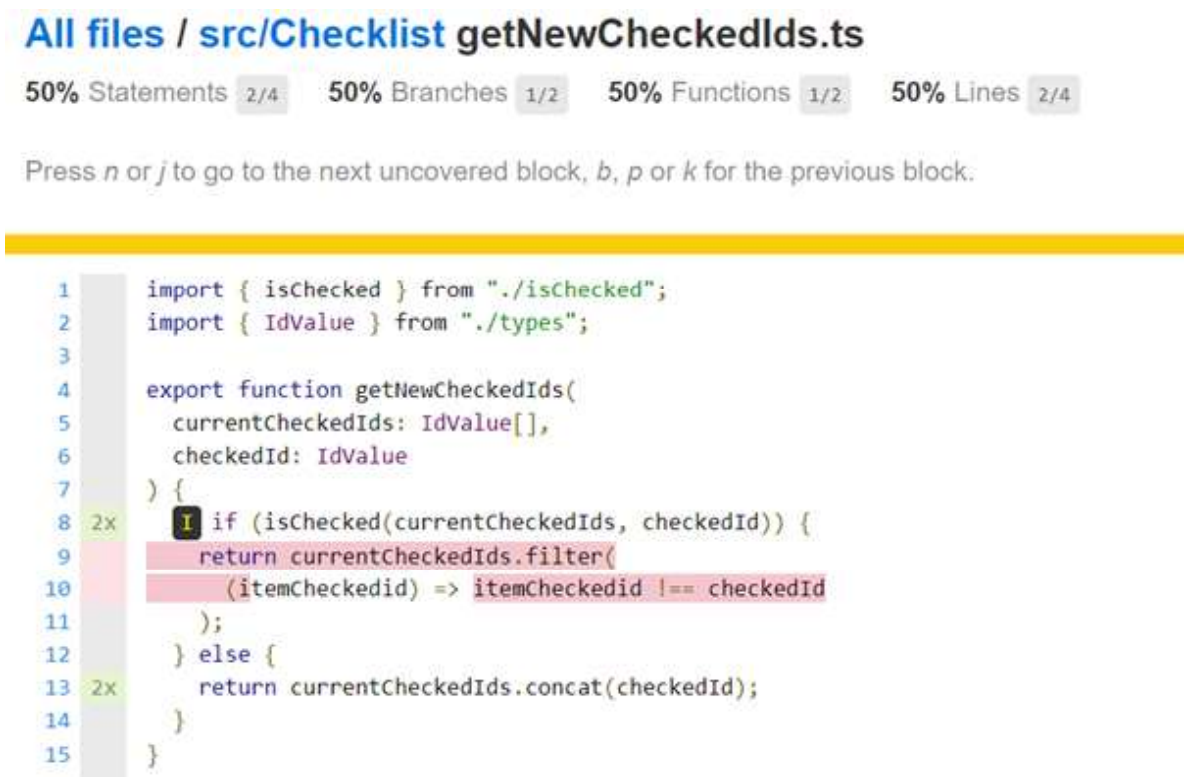


Figure 12.12 – Coverage report for getNewCheckedIds.ts

We can see that the uncovered lines 9 and 10 are very clearly highlighted in the `getNewCheckedIds.ts` file.

So, the HTML coverage report is useful in a large code base because it starts with high-level coverage and allows you to drill into coverage on specific folders and files. When viewing a file in the report, we can quickly determine where the uncovered code is because it is clearly highlighted.

Next, we will update our tests so that lines 9 and 10 in `getNewCheckedIds.ts` are covered.

Gaining full coverage on the checklist component

The logic not currently being checked by tests is the logic used when a list item is clicked but has already been checked. We will extend the `'should check items when clicked'` test to cover this logic. Carry out the following steps:

1. Open `Checklist.test.tsx` and rename the `'should check items when clicked'` test as follows:

```
test('should check and uncheck items when clicked', async
() => {
  ...
});
```

2. Add the following highlighted lines at the end of the test to click the checkbox for a second time and check it is unchecked:

```
test('should check and uncheck items when clicked', async
() => {
  const user = userEvent.setup();
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role: 'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
    />
  );
  const lucyCheckbox = screen.getByTestId(
    'Checklist__input__1'
  );
  expect(lucyCheckbox).not.toBeChecked();
  await user.click(lucyCheckbox);
  expect(lucyCheckbox).toBeChecked();
  await user.click(lucyCheckbox);
  expect(lucyCheckbox).not.toBeChecked();
});
```

3. In the terminal, rerun the tests with coverage:

```
npm run test -- --coverage --watchAll=false
```

All the tests still pass, and the coverage on the checklist component is now reported as 100% on all the statistics:

```
-----------------------------|---------|----------|---------|---------|-------------------
File                         | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------------|---------|----------|---------|---------|-------------------
All files                    |    61.7 |    66.66 |   66.66 |   62.22 |
 src                         |       0 |        0 |       0 |       0 |
  App.tsx                    |       0 |        0 |       0 |       0 | 5-14
  index.tsx                  |       0 |      100 |     100 |       0 | 7-19
  reportWebVitals.ts         |       0 |        0 |       0 |       0 | 3-10
 src/Checklist               |     100 |      100 |     100 |     100 |
  Checklist.tsx              |     100 |      100 |     100 |     100 |
  assertValueCanBeRendered.ts|     100 |      100 |     100 |     100 |
  getNewCheckedIds.ts        |     100 |      100 |     100 |     100 |
  index.ts                   |       0 |        0 |       0 |       0 |
  isChecked.ts               |     100 |      100 |     100 |     100 |
  types.ts                   |       0 |        0 |       0 |       0 |
  useChecked.ts              |     100 |      100 |     100 |     100 |
-----------------------------|---------|----------|---------|---------|-------------------
```

Figure 12.13 – 100% coverage on the checklist component

The checklist component is now well covered. However, it is a little annoying that `index.ts` and `types.ts` appear in the report with zero coverage. We'll resolve this next.

Ignoring files in the coverage report

We will remove `index.ts` and `types.ts` from the coverage report because they don't contain any logic and create unnecessary noise. Carry out the following steps:

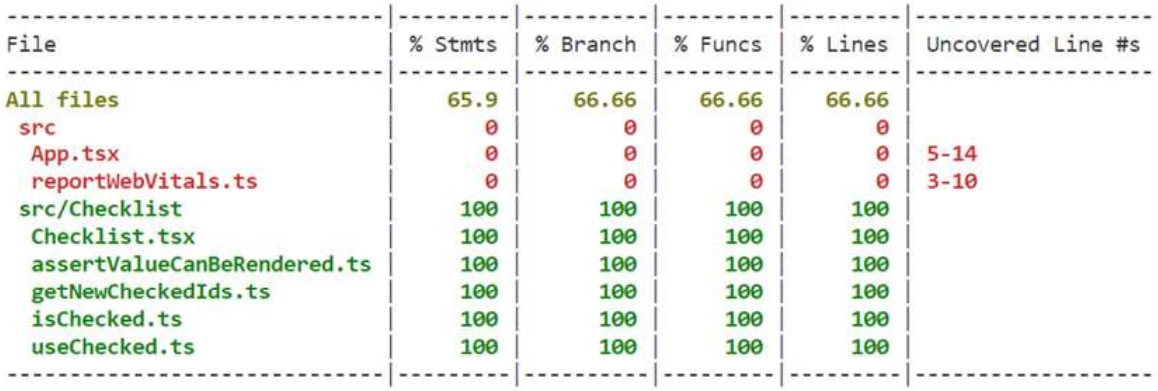
1. Open the `package.json` file. We can configure Jest in the `package.json` file in a `jest` field, and there is a `coveragePathIgnorePatterns` configuration option for removing files from the coverage report. Add the following Jest configuration to `package.json` to ignore the `types.ts` and `index.ts` files:

```
{
  ...,
  "jest": {
    "coveragePathIgnorePatterns": [
      "types.ts",
      "index.ts"
    ]
  }
}
```

2. In the terminal, rerun the tests with coverage:

```
npm run test -- --coverage --watchAll=false
```

The `types.ts` and `index.ts` files are removed from the coverage report:



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	65.9	66.66	66.66	66.66	
src	0	0	0	0	
App.tsx	0	0	0	0	5-14
reportWebVitals.ts	0	0	0	0	3-10
src/Checklist	100	100	100	100	
Checklist.tsx	100	100	100	100	
assertValueCanBeRendered.ts	100	100	100	100	
getNewCheckedIds.ts	100	100	100	100	
isChecked.ts	100	100	100	100	
useChecked.ts	100	100	100	100	

Figure 12.14 – types.ts and index.ts files removed from the coverage report

That completes this section on code coverage. Here's a quick recap:

- The `--coverage` option outputs a code coverage report after the tests have run.
- An interactive HTML code coverage report is generated in addition to the one in the terminal. This is useful on a large test suite to drill into uncovered code.
- Both report formats highlight uncovered code, giving us valuable information to improve our test suite.

Summary

In this chapter, we created tests on a checklist component using Jest and React Testing Library. In addition, we learned about common Jest matchers in Jest's core package and useful matchers for component testing in a companion package called `jest-dom`.

We used Jest's test runner and used options to run certain tests. This is particularly useful on large code bases.

We learned about the wide variety of queries available in React Testing Library to select elements in different ways. We used the `getByText` query extensively in the checklist tests. We also created a test ID on list item checkboxes so that the `getByTestId` query could be used to select them uniquely.

We learned that the `user-event` package is an excellent way of simulating user interactions that are decoupled from the implementation. We used this to simulate a user clicking a list item checkbox.

We learned how to produce code coverage reports and understood all the statistics in the report. The report included information about uncovered code, which we used to gain 100% coverage on the checklist component.

So, we have reached the end of this book. You are now comfortable with both React and TypeScript and have excellent knowledge in areas outside React core, such as styling, client-side routing, forms, and web APIs. You will be able to develop components that are reusable across different pages and even different apps. On top of that, you will now be able to write a robust test suite so that you can ship new features with confidence.

In summary, the knowledge from this book will allow you to efficiently build the frontend of large and complex apps with React and TypeScript. I hope you have enjoyed reading this book as much as I did writing it!

Questions

Answer the following questions to check what you have learned in this chapter:

1. We have written some tests for a `HomePage` component and placed them in a file called `HomePage.tests.tsx`. However, the tests aren't run when the `npm test` command is executed—not even when the *a* key is pressed to run all the tests. What do you think the problem might be?
2. Why doesn't the following expectation pass? How could this be resolved?

   ```
   expect({ name: 'Bob' }).toBe({ name: 'Bob' });
   ```

3. Which matcher can be used to check that a variable isn't `null`?
4. Here's an expectation that checks whether a **Save** button is disabled:

   ```
   expect(
     screen.getByText('Save').hasAttribute('disabled')
   ).toBe(true);
   ```

   The expectation passes as expected, but is there a different matcher that can be used to simplify this?
5. Write a test for the `getNewCheckedIds` function we used in this chapter. The test should check if an ID is removed from the array of checked IDs if it is already in the array.
6. We have a `form` element containing a **Save** button only when data has been loaded into fields from a server API. We have used the `findBy` query type so that the query retries until the data has been fetched:

   ```
   expect(screen.findByText('Save')).toBeInTheDocument();
   ```

   However, the expectation doesn't work—can you spot the problem?

7. The following expectation attempts to check that a **Save** button isn't in the DOM:

```
expect(screen.getByText('Save')).toBe(null);
```

This doesn't work as expected, though. Instead, an error is raised because the **Save** button can't be found. How can this be resolved?

Answers

1. The problem is that the file extension is `tests.tsx` rather than `test.tsx`.
2. The `toBe` matcher should only be used for checking primitive values such as numbers and strings—this is an object. The `toStrictEqual` matcher should be used to check objects because it checks the values of all its properties instead of the object reference:

```
expect({ name: 'Bob' }).toStrictEqual({ name: 'Bob' });
```

3. The `not` and `toBeNull` matchers can be combined to check a variable isn't `null`:

```
expect(something).not.toBeNull();
```

4. The `toBeDisabled` matcher can be used from `jest-dom`:

```
expect(screen.getByText('Save')).toBeDisabled();
```

5. Here's a test:

```
test('should remove id when already in checked ids', ()
=> {
  const result = getNewCheckedIds([1, 2, 3], 2);
  expect(result).toStrictEqual([1, 3]);
});
```

6. The `findBy` query type requires awaiting because it is asynchronous:

```
expect(await screen.findByText('Save')).
toBeInTheDocument();
```

7. The `queryBy` query type can be used because it doesn't throw an exception when an element isn't found. In addition, the `not` and `toBeInTheDocument` matchers can be used to check that the element isn't in the DOM:

```
expect(screen.queryByText('Save')).not.
toBeInTheDocument();
```