Creating strongly-typed function component state with useReducer

In this lesson, we are going to learn how to create strongly-typed state with the 'useReducer' hook.

WE'LL COVER THE FOLLOWING

Implementing a basic useReducer

The `useReducer` hook is excellent for implementing complex state scenarios. It uses a Redux like pattern for state management where object literals called *actions* are passed into a function called a *reducer* to change state.

We are going to leverage `useReducer` to manage the count state within a `Counter` component. Open the CodeSandbox project by clicking the link below and let's get started:

CodeSandbox useReducer starter project

First, we are going to create the action types that will be referred to in the reducer we will eventually create. We are going to have two actions for incrementing and decrementing the counter. Let's create these below the `State` type:

```
type Increment = {
  readonly type: 'increment';
  readonly incrementStep: number;
};
type Decrement = {
  readonly type: 'decrement';
  readonly decrementStep: number;
```

```
};
```

So, the actions will have a `type` property, which uniquely defines the type of action that needs to be performed along with the amount that the counter will need to be incremented or decremented. We have used the `readonly` keyword on the action properties so that they are immutable.

Is it possible to create these action types using an interface rather than a type alias?

Show Answer

Next, we are going to create a union type called `Actions` from the `Increment` and `Decrement` types. This union type will eventually be used as the type for a parameter for the action in the reducer. Let's add this under the `Decrement` type:

```
type Actions = Increment | Decrement;
```

Moving on to the reducer function. Let's create the function's signature:

```
const reducer = (state: State, action: Actions): State => {

};
```

The function takes in two parameters for the current state and the action that needs to be performed to update the state.

Next, we'll use a `switch` statement to handle each type of action, returning the new state in each branch:

```
const reducer = (state: State, action: Actions): State => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + action.incrementStep };
    case 'decrement':
      return { count: state.count - action.decrementStep };
  }
  return state;
};
```

We have also returned the original state at the bottom of the function. This line should never be reached, but it keeps TypeScript happy with the function's return type.

What is the type of the `action` parameter when inside the *'increment'* `switch` branch?

Show Answer

We are now going to make the reducer super type-safe by leveraging the `never` type. First, we will understand and expose the problem that the `never` will eventually solve. Let's pretend time has moved on, and the reducer needs to cater to a new action, *double*. Let's add this action type beneath the `Decrement` type:

```
type Double = {
  readonly type: 'double';
};
```

Let's add this to the `Actions` union type as well:

```
type Actions = Increment | Decrement | Double;
```

Wouldn't it be nice if the editor reminded us to implement the additional `switch` branch in the reducer? TypeScript knows that there is an action type that we haven't handled, so maybe this is possible? Well, it certainly is! We can use the `never` type in the `switch` statement's `default` branch to tell the TypeScript compiler that this should never be reached:

```
const reducer = (state: State, action: Actions): State => {
  switch (action.type) {
    ...
    default:
      neverReached(action);
  }
  return state;
};
```

```
const neverReached = (never: never) => {};
```

After we have made this change to the reducer, the editor informs us that the `default` `switch` branch will be reached if the `Double` action is passed in.

If we add a `switch` branch for `'double'` action type the compilation error goes away:

```
const reducer = (state: State, action: Actions): State => {
  switch (action.type) {
    ...
    case 'double':
      return { count: state.count * 2 };
    default:
      neverReached(action);
  }
  return state;
};
```

Now that we have fully implemented the reducer, we can use it within the `Counter` component using the `useReducer` hook.

```
const Counter = ( ... ) => {
  const [state, dispatch] = React.useReducer(reducer, { count: initialCoun
t });

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

We pass the reducer function into the first parameter and the initial state into the second parameter of the `useReducer` hook. This hook returns the current state and a dispatch function that we can use to start the process of updating the state.

What is the type of the destructured `state` variable and `dispatch` function from the `useReducer` hook? Hover over them in the CodeSandbox project and find out.

Show Answer

TypeScript has cleverly inferred these types. We can explicitly define the types

TypeScript has cleverly inferred these types. We can explicitly define the types by passing them into the generic parameters of `useReducer`:

```
const Counter = ( ... ) => {
  const [state, dispatch] = React.useReducer<React.Reducer<State, Actions>
>(
    reducer,
    { count: initialCount }
  );

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

We can complete our first implementation of the `Counter` component by rendering the current count and dispatching the actions when the buttons are clicked:

```
const Counter = ( ... ) => {
  ...
  return (
    <div>
      <div>{state.count}</div>
      <button onClick={() => dispatch({ type: "increment", incrementStep
})}>
        Add {incrementStep}
      </button>
      <button onClick={() => dispatch({ type: "decrement", decrementStep
})}>
        Subtract {decrementStep}
      </button>
      <button onClick={() => dispatch({ type: "double" })}>
        Double
      </button>
    </div>
  );
};
```

Give the app a try, the buttons should update the counter.

Specifying the initial state via a function

The initial state in the `Counter` component is directly passed into `useReducer` at the moment. We have the option to define this in a function which is useful if there are a few steps that derive the initial state. This is also useful if the

function is called by another action such as the resetting state. We are going to add a *Reset* button to the `Counter` component and leverage this way of calling `useReducer`.

Let's start by adding the *Reset* button:

```
const Counter = ( ... ) => {
  ...
  return (
    <div>
      ...
      <button onClick={() => dispatch({ type: "reset", initialCount })}>
        Reset
      </button>
    </div>
  );
};
```

We have referenced an action type, `"reset"`, that doesn't exist yet, so a compilation error shows. Let's add this action type now and add it to the `Actions` type as well:

```
type Reset = {
  readonly type: "reset";
  readonly initialCount: number;
};
type Actions = Increment | Decrement | Double | Reset;
```

We are now reminded to handle this action type in the reducer by an editor. Let's do this:

```
const reducer = (state: State, action: Actions): State => {
  switch (action.type) {
    ...
    case "reset":
      return resetState(action.initialCount);
    default:
      neverReached(action);
  }
  return state;
};
```

`resetState` is the function that will return the state with the initial value. Let's add this just below the `State` type definition:

```
const resetState = (initialCount: number): State => ({
  count: initialCount
});
```

We can now use this `resetState` in the `useReducer` function to define the initial state value:

```
const [state, dispatch] = React.useReducer(reducer, initialCount, resetSta
te);
```

... or if we want to pass the types rather than have them inferred explicitly:

```
const [state, dispatch] = React.useReducer<
  React.Reducer<State, Actions>,
  number
>(reducer, initialCount, resetState);
```

That completes the implementation. Give it a try! Don't forget to try passing an `initialCount` to the `Counter` to check that this works:

```
const rootElement = document.getElementById("root");
render(
  <Counter incrementStep={1} decrementStep={2} initialCount={5} />,
  rootElement
);
```

Wrap up

That concludes this lesson. If we explicitly define the return type for the state on the reducer, the `useReducer` hook will usually infer the state type correctly. Also, if we specify a type for the `action` parameter in the reducer, the `useReducer` hook generally infers the type of the `dispatch` function correctly as well. If TypeScript fails to infer the types correctly, we can pass them in as generic parameters to the `useReducer` hook.

We are now starting to feel the benefit of TypeScript with React, reminding us where code is missed or incorrect in our implementation.

Next, let's double-check what we have learned from the last couple of lessons

with a quiz.