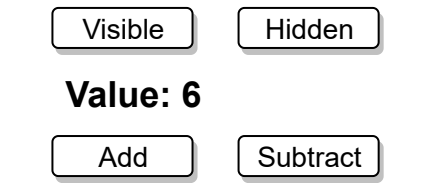
# Simple Counter App

This lesson uses all we have learned about Redux in the previous few lessons to make a Simple Counter Application. Let's dive in!

WE'LL COVER THE FOLLOWING



Let’s create a simple counter app that uses all the APIs above.




Visual: Simple Counter App

The “Add” and “Subtract” buttons will change a value in our store. “Visible” and “Hidden” will control its visibility.

## Modeling the actions #

For me, every Redux feature starts with modeling the action types and defining what state we want to keep. In our case, we have three operations going on - adding, subtracting and managing visibility. So we will go with the following:

```
const ADD = 'ADD';
const SUBTRACT = 'SUBTRACT';
const CHANGE_VISIBILITY = 'CHANGE_VISIBILITY';

const add = () => ({ type: ADD });
const subtract = () => ({ type: SUBTRACT });
```

```
const changeVisibility = visible => ({
  type: CHANGE_VISIBILITY,


  visible
});
```

## Store and its reducers #

There is something that we didn't talk about while explaining the store and reducers. We usually have more than one reducer because we want to manage multiple things. The store is one though and we in theory have only one state object. What happens in most of the apps running in production is that the application state is a composition of slices. Every slice represents a part of our system. In this very small example we have counting and visibility slices. So our initial state looks like this:

```
const initialState = {
  counter: {
    value: 0
  },
  visible: true
};
```

We must define separate reducers for both parts. This is to introduce some flexibility and to improve the readability of our code. Imagine if we have a giant app with ten or more state slices and we keep working within a single function. It will be too difficult to manage.

Redux comes with a helper that allows us to target a specific part of the state and assign a reducer to it. It is called `combineReducers`:

```
import { createStore, combineReducers } from 'redux';

const rootReducer = combineReducers({
  counter: function A() { ... },
  visible: function B() { ... }
});
const store = createStore(rootReducer);
```

Function `A` receives only the `counter` slice as a state and needs to return only that part. The same goes for `B`, it also receives only the `counter` slice as a state. Accepts a boolean (the value of `visible`) and must return a boolean.

The reducer for our counter slice should take into account both actions `ADD,`

`SUBTRACT` and based on them calculate the new `counter` state.

```
const counterReducer = function (state, action) {
  if (action.type === ADD) {
    return { value: state.value + 1 };
  } else if (action.type === SUBTRACT) {
    return { value: state.value - 1 };
  }
  return state || { value: 0 };
};
```

Every reducer is fired at least once when the store is initialized. In that very first run the `state` is `undefined` and the `action` is `{ type: "@@redux/INIT"}`. In this case our reducer should return the initial value of our data - `{ value: 0 }`.

The reducer for the visibility is pretty similar except that it waits for `CHANGE_VISIBILITY` action:

```
const visibilityReducer = function (state, action) {
  if (action.type === CHANGE_VISIBILITY) {
    return action.visible;
  }
  return true;
};
```

And at the end we have to pass both reducers to `combineReducers` so we create our `rootReducer`.

```
const rootReducer = combineReducers({
  counter: counterReducer,
  visible: visibilityReducer
});
```

## Selectors #

Before moving to the React components we have to mention the concept of a *selector*. From the previous section we know that our state is usually divided into different parts. We have dedicated reducers to update the data but when it comes to fetching it we still have a single object. Here is the place where the selectors come in handy. The selector is a function that accepts the whole state object and extracts only the information that we need. For example in our small app we need two of those:

---

```
const getCounterValue = state => state.counter.value;
const getVisibility = state => state.visible;
```

A counter app is too small to see the real power of writing such helpers. However, for a big project, it is quite different. And it is not just about saving a few lines of code. Neither is it about readability. Selectors come with this stuff but they are also contextual and may contain logic. Since they have access to the whole state they are able to answer business logic related questions. Like for example “Is the user authorized to do X while being on page Y”. This may be done in a single selector.

## React components #

Let’s first deal with the UI that manages the visibility of the counter.

```
function Visibility({ changeVisibility }) {
  return (
    <div>
      <button onClick={ () => changeVisibility(true) }>
        Visible
      </button>
      <button onClick={ () => changeVisibility(false) }>
        Hidden
      </button>
    </div>
  );
}

const VisibilityConnected = connect(
  null,
  dispatch => ({
    changeVisibility: value => dispatch(changeVisibility(value))
  })
)(Visibility);
```

We have two buttons `Visible` and `Hidden`. They both fire `CHANGE_VISIBILITY` action but the first one passes `true` as a value while the second one `false`. The `VisibilityConnected` component class gets created as a result of the wiring done via Redux’s `connect`. Notice that we pass `null` as `mapStateToProps` because we don’t need any of the data in the store. We just need to `dispatch` an action.

The second component is slightly more complicated. It is named `Counter` and renders two buttons and the counter value.

```
function Counter({ value, add, subtract }) {
```

```
  return (
    <div>
      <p>Value: { value }</p>

      <button onClick={ add }>Add</button>
      <button onClick={ subtract }>Subtract</button>
    </div>
  );
}

const CounterConnected = connect(
  state => ({
    value: getCounterValue(state)
  }),
  dispatch => ({
    add: () => dispatch(add()),
    subtract: () => dispatch(subtract())
  })
)(Counter);
```

We now need both `mapStateToProps` and `mapDispatchToProps` because we want to read data from the store and dispatch actions. Our component receives three props - `value`, `add` and `subtract`.

The very last bit is an `App` component where we compose the application.

```
function App({ visible }) {
  return (
    <div>
      <VisibilityConnected />
      { visible && <CounterConnected /> }
    </div>
  );
}
const AppConnected = connect(
  state => ({
    visible: getVisibility(state)
  })
)(App);
```

We again need to `connect` our component because we want to control the visibility of the counter. The `getVisibility` selector returns a boolean that indicates whether `CounterConnected` will be rendered or not.

---

In the next and final lesson of this chapter, we will present some final thoughts and executable code of Simple Counter App