

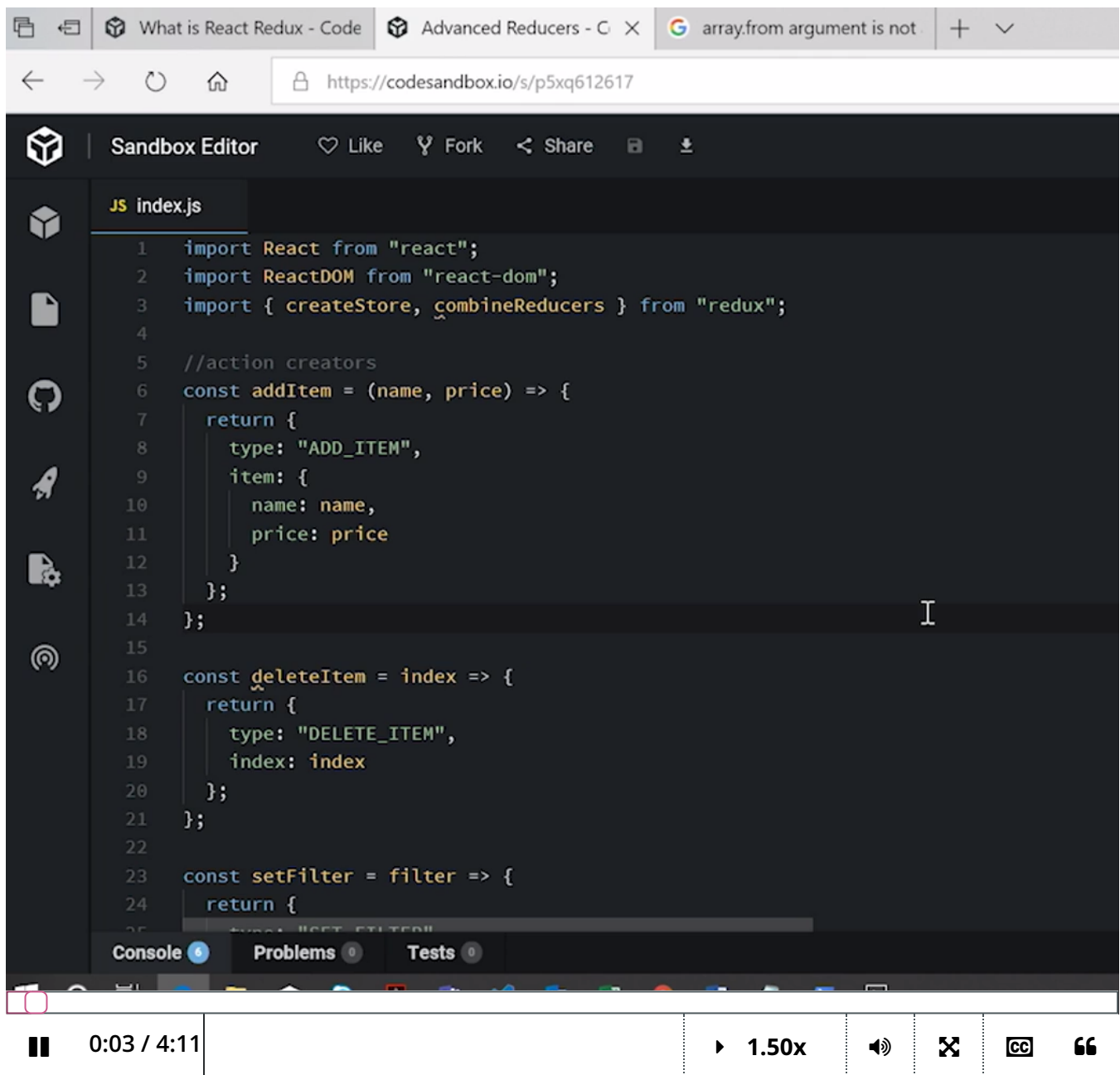


[Course](#) > [Redux](#) > [Advanc...](#) > [Advanc...](#)

Audit Access Expires Jun 20, 2020

You lose all access to this course, including your progress, on Jun 20, 2020.

Advanced Reducers



The screenshot shows a video player interface. The top part of the video displays a web browser with the URL `https://codesandbox.io/s/p5xq612617`. Below the browser is a 'Sandbox Editor' window. The editor has a sidebar on the left with icons for file explorer, GitHub, and other tools. The main area of the editor shows a JavaScript file named `index.js` with the following code:

```
1 import React from "react";
2 import ReactDOM from "react-dom";
3 import { createStore, combineReducers } from "redux";
4
5 //action creators
6 const addItem = (name, price) => {
7   return {
8     type: "ADD_ITEM",
9     item: {
10       name: name,
11       price: price
12     }
13   };
14 };
15
16 const deleteItem = index => {
17   return {
18     type: "DELETE_ITEM",
19     index: index
20   };
21 };
22
23 const setFilter = filter => {
24   return {
25     type: "SET_FILTER",
```

At the bottom of the video player, there is a progress bar showing `0:03 / 4:11`, a play button, a speed control set to `1.50x`, and icons for volume, full screen, and subtitles.

Video

[Download video file](#)

Transcripts

[Download SubRip \(.srt\) file](#)

[Download Text \(.txt\) file](#)

Advanced Reducers

Advanced Reducers

As your application grows to have more and more actions and an increasingly complicated state tree, it becomes important to split up your reducers into multiple reducers that handle separate parts of your application state.

In this lesson, we will be building a more complicated reducer for a more complicated shopping cart application. User's should be able to add and remove items from their shopping cart but this time the items will have prices as well as product names. User's should also have the option to sort the shopping cart items by alphabetical order or by price. Lastly, user's should also be able to use a coupon to get certain percentage discount off all items.

Designing the actions

We must first design the actions before we can design the reducers. There are four distinct actions that a user can take:

- Add item to the cart
- Delete item from the cart
- Set sort by filter
- Set coupon discount percentage

We can turn these user scenarios into the following actions:

```
//ADD_ITEM
{
  type: 'ADD_ITEM',
  item: {
    name: string,
    price: double
  }
}

//DELETE_ITEM
{
  type: 'DELETE_ITEM',
  index: integer //index value
}

//SET_FILTER
{
  type: 'SET_FILTER',
  filter: string //either none, alphabetical or price
}

//SET_DISCOUNT
{
  type: 'SET_DISCOUNT',
  discount: double //percentage amount
}
```

Designing the state tree

We must next design the state tree before we can design the reducers. The state tree should be one giant object that contains an array for the shopping cart items and attributes to hold the values for the filter type and discount amount.

State tree:

```
{  
  items: [],  
  filter: value, //either none, alphabetical or price,  
  discount: value //percentage amount  
}
```

It is important to only keep track of necessary data. There is no need to add an additional state attribute, if you can already calculate it from the existing state. For example, there shouldn't be a discounted items array since we can calculate that from the items array and the discount attribute.

Designing the reducers

Let's start by trying to create one giant reducer. In the process, you will see why it is a better idea to split it up into smaller reducers.

```
var initialState = {
  items: [],
  filter: 'none'
  discount: 0
}

const giantReducer = (state = initialState, action) =>{
  switch(action.type){
    case 'ADD_ITEM':
      return Object.assign({},state,
        items: [...state.items, action.item ]
      )
    case 'DELETE_ITEM':
      return Object.assign({},state,
        items: [...state.slice(0,action.index),...state.slice(action
      )
    case 'SET_FILTER':
      return Object.assign({},state,
        filter: action.filter
      )
    case 'SET_DISCOUNT':
      return Object.assign({},state,
        discount: action.discount
      )
  }
}
```

It was a bit annoying to have to use `Object.assign()` to generate the entire state tree when we only needed to modify one state attribute per action. It will also only get more difficult as we add more actions and as our state tree gets more complicated. Luckily, we can split the reducer up into several reducers that each handle one attribute of the state tree.

We can break out `ADD_ITEM` and `DELETE_ITEM` into their own reducer because they both only affect the `items` attribute. It makes it a bit easier to maintain since we can just return an array instead of the whole state tree:

```
//items reducer
const items = (state = [], action) => { //notice default state is now []
  switch(action.type){
    case 'ADD_ITEM':
      return [...state.items, action.item ]
    })
    case 'DELETE_ITEM':
      return [...state.slice(0,action.index),...state.slice(action.index+1,state.length)]
    })
    default:
      return state
  }
}
```

We can also break out SET_FILTER and SET_DISCOUNT into their separate reducers as well:

```
//items reducer
const filter = (state = [], action) => { //notice default state is now []
  switch(action.type){
    case 'ADD_ITEM':
      return action.filter
    default:
      return state
  }
}

const discount= (state = 'none', action) => { //notice default state is now 'none'
  switch(action.type){
    case 'SET_DISCOUNT':
      return action.discount
    default:
      return state
  }
}
```

We can combine these reducers together by returning an object and placing the child reducers next to the attributes they belong to. The small reducers will return the new state for the attribute they are in charge of and then the encompassing reducer will group

them all together. We also no longer need an initial state for the encompassing reducer as long as the all of the child reducers have initial states. It is a good practice to have a separate reducer for each top level attribute in your state tree.

```
const giantReducer = (state, action) =>{ //don't need an initial sta
  {
    items: items(state.items,action),
    filter: filter(state.filter,action),
    discount: discount(state.discount,action)
  }
}
```

Redux even has a `combineReducers` method that does the combining for us!

The `combineReducers()` method does the same thing as the example above:

```
const reducer = combineReducers({
  items,
  filter,
  discount,
})
```

You will notice that the reducer name ends up being the attribute name in the state. If you want to differ the reducer name and the attribute names you can do the following:

```
const reducer = combineReducers({
  a : items,
  b : filter,
  c : discount,
})
```

The result of this will be the same as the following:


```
const giantReducer = (state, action) =>{
  {
    a: items(state.a,action),
    b: filter(state.b,action),
    c: discount(state.c,action)
  }
}
```

Importing Reducers from a separate file

Its a common practice to create all of your reducers in a separate file and then import them into your main file to be combined with `combineReducers()`.

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

import * as reducers from './reducers'

const reducer = combineReducers(reducers)
var store = createStore(reducer)
```

Try out this live example on CodeSandbox: <https://codesandbox.io/s/p5xq612617>

© All Rights Reserved