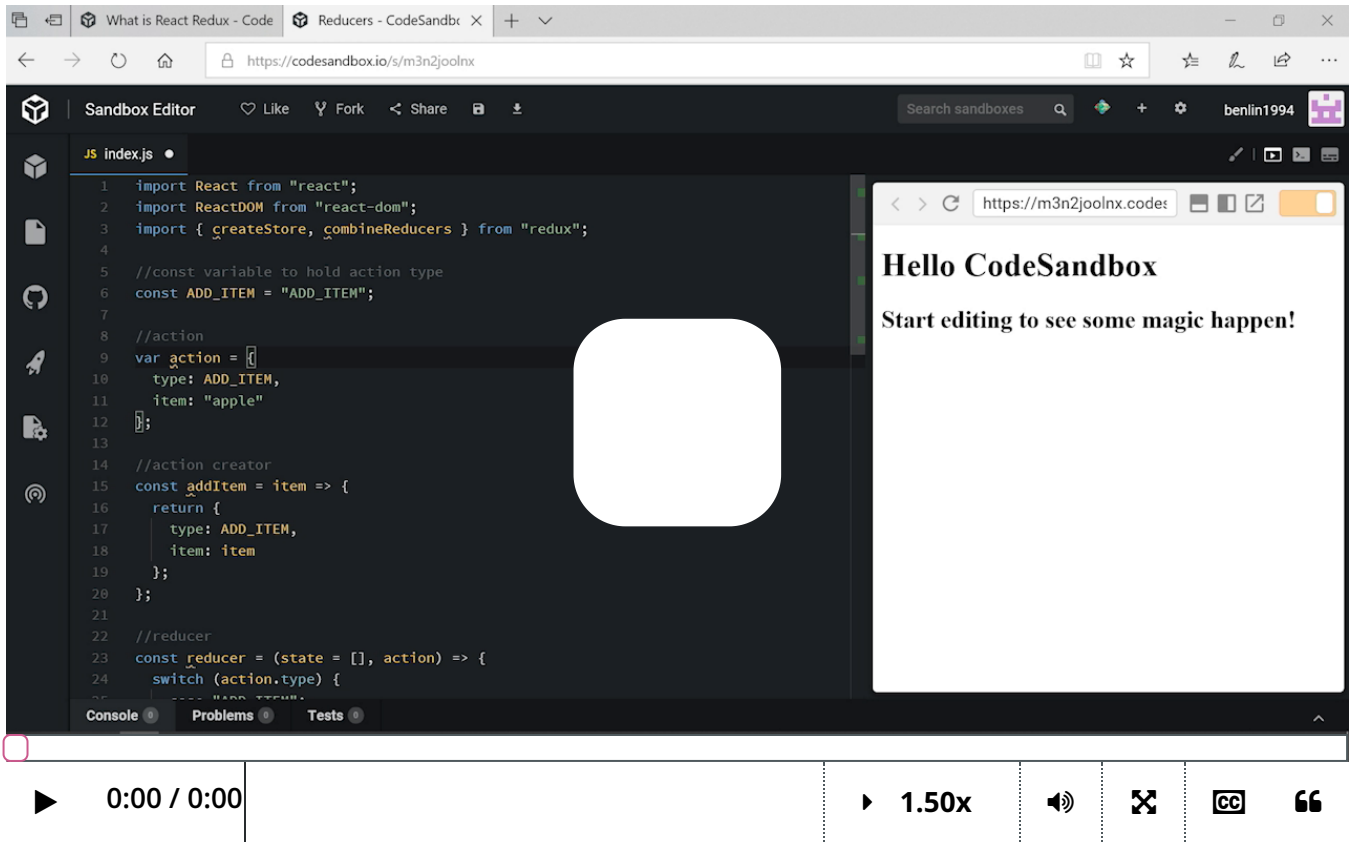
Audit Access Expires Jun 20, 2020
You lose all access to this course, including your progress, on Jun 20, 2020.

Reducers



Video

Download video file

Transcripts

Download SubRip (.srt) file

Download Text (.txt) file

Reducers

Reducers

Reducers are just functions that define how the state should change after an action is dispatched. In this lesson we will be building a Reducer for a simple shopping cart application.

Reducers must be pure functions

A reducer must be a pure function that takes in the previous state and a dispatched action as inputs and returns the new state as an output.

```
const items = (state, action ) => {
 //return new state based on previous state and action
}
```

A pure function should constantly return the same output when given the same input. There should never be any API calls, calls to Math.random() or Date.now(), mutations to the previous state, or any other surprises. It should just be a functional calculation. Redux triggers changes to React components whenever there is a new state object loaded into the store. So if you mutate the old state object instead of making a new one, Redux won't see the change and will not update the React components.

State shape and initial state

It is important to understand the shape of the state tree before coding up a reducer. Otherwise, how will we know how to update the state if we don't know how the state is supposed to be organized. In the shopping cart example, the shape tree will look like the following array:

```
  ['apple','banana','etc...']
```

The state argument will be undefined the first time our reducer is called, so it is important to provide a default initial state. We can do so by using the following syntax:

```
const items = (state = [], action ) => {
 //return new state based on previous state and action
}
```

Handling Actions

We can use a switch statement to handle different types of actions:

```
const items = (state = [], action) => { //providing an empty array a
  switch(action.type){
    case 'ADD_ITEM':
      //return new state with added item
    case 'DELETE_ITEM':
      //return new state with deleted item
    default:
      return state //otherwise, just return the old state
  }
}
```

We can then implement the logic to update the state based on the dispatched action.

Implementing ADD_ITEM

The ADD_ITEM action will look like the following:

ADD_ITEM:

```
{
  type: 'ADD_ITEM',
  item: 'apple' //any string value
}
```

To implement ADD_ITEM, we can generate a new array, load in the previous state's items using the spread operator, and then add in the new item from the action.

```
    case 'ADD_ITEM':
      return [...state, action.item]
```

Implementing DELETE_ITEM

The DELETE_ITEM action will look like the following:

DELETE_ITEM:

```
{
  type: 'DELETE_ITEM',
  index: 0 //any index number
}
```

To implement DELETE_ITEM, we can generate a new array, load in the previous state's items up to the removed index, and then load in the previous state's items that come after the removed index.

```
    case 'DELETE_ITEM':
      return [...state.slice(0,action.index),...state.slice(action.i
```

Handling invalid actions

Invalid actions can be caught with the default case of the switch statement, which will just return the original state.

```
    default:
      return state //otherwise, just return the old state
```

Helpful methods for pure functions

There are several JavaScript methods that are helpful when writing pure functions. You should take a closer look at the following functions:

- array.slice() - used for returning a new array that contains a portion of an existing array

```
var array = [0,1,2,3,4,5,6]
var newArray = array.slice(0,5)
console.log(newArray)
// [0,1,2,3,4]
```

- Object.assign - used for creating a new object and merging over other objects into the new object

```
var oldState = { id: 123, value: 'abc'}
var newState = Object.assign({}, oldObject, {value:'new value'})
console.log(newState)
// {id: 123, value: 'new value'}
```

- ... (spread operater) - used for loading in data from an existing object or array

```
var oldState = { id: 123, value: 'abc'}
var newData = {value:'new value'}
var newState = {...oldObject, ...newData}
console.log(newState)
// {id: 123, value: 'new value'}
var array = [0,1,2,3,4]
var newArray = [...array, 5, 6]
console.log(newArray)
//[0,1,2,3,4,5,6]
```

Try out this live example on CodeSandbox: https://codesandbox.io/s/m3n2joolnx