# STRATEGIES FOR COMPLEX STATE

## Introduction to Strategies for Complex State

In the last lesson, you built a simple counter app whose store state was just a single number. Though the counter app illustrates how Redux can manage the state of an application, it isn't a great example of an application that needs Redux.

Redux really shines when used in applications with many features and a lot of data where having a centralized store to keep it all organized is advantageous. In this lesson, you will learn strategies for managing an application with a more complex store state and, in the process, you will begin to build an app that will grow throughout the rest of this course.

In the browser, you can see the final product. This application, which we will refer to as the "Recipes App", does the following:

- displays a set of recipes which are pulled from a database.
- allows the user to add/remove their favorite recipes to/from a separate list.
- allows the user to enter a search term to filter the visible recipes.

Now, imagine you are working for the software development company whose main product is this Recipes application. The product manager has determined the desired features and functionality, the graphic designer has defined its style, and the React engineer has created its components. Now it is up to you, the Redux Engineer, to design the state management system that can put it all together!

*In reality, the Front-End Engineer would implement both React and Redux.*

Before continuing on, make sure that you are familiar with the following terms and concepts relating to React and Redux:

- React
  - How to create components
  - How to render components using `ReactDOM.render()`
  - How to nest components and pass data through props
- Redux
  - One-way data flow model: State → View → Actions → State → View ...

- How to create a reducer function: `(state, action) => nextState`
- How to write action objects and action creators
- How to create a `store` using `createStore()`
- How to use the `store` methods `getState()`, `dispatch()`, and `subscribe()`

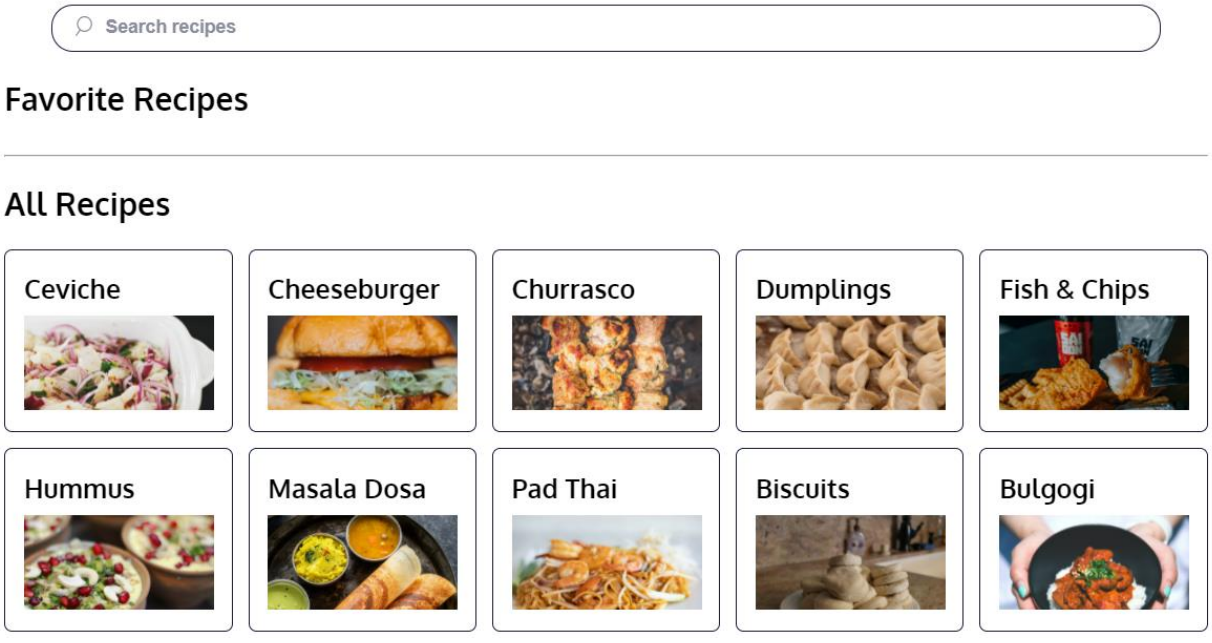
*Note to learners: The slightly wordy nature of Redux means that the examples in this course can be quite large. It is recommended that you expand the "Learn" section while reading through this lesson's materials. You can do so by clicking-and-dragging the dividing line that separates the "Learn" section from the code editor.*

## Instructions

Spend a few moments getting familiar with the features of this application. While you're at it, consider the following:

- What React components exist in this application?
- What data does each component need from the store?
- What actions occur within this application?
- How do those actions update the store's state?

Throughout the rest of this course you will be designing the store's state structure, creating action creators to describe state changes, writing a reducer to execute state changes, and connecting the Redux store to the existing React components. Let's begin!



**Slices**

Redux is best suited for complex applications with many features that each have some state-related data to be managed. In these cases, objects are the go-to data type to represent the entire store's state.

For example, consider a todo app that allows a user to:

- add to a list of todos
- mark individual todos as complete or incomplete
- apply a filter to show only the completed todos, only the incomplete todos, or all of the todos

After adding a few todos and setting the filter to show incomplete todos, the state might look like this:

```
state = {
  todos: [
    {
      id: 0,
      text: 'Complete the Learn Redux course',
      isCompleted: false
    },
    {
      id: 1,
      text: 'Build a counter app',
      isCompleted: true
    },
  ],
  visibilityFilter: 'SHOW_INCOMPLETE'
};
```

In a Redux application, the top-level `state` properties, `state.todos` and `state.visibilityFilter`, are known as *slices*. Each slice typically represents a different feature of the entire application. Notice that a slice can be any data value, like an array of objects (`state.todos`) or just a string (`state.visibilityFilter`).

As a best practice, most Redux applications begin with an `initialState` that allows the programmer to do two key things:

1. Plan out the general structure of the state
2. Provide an initial state value to the reducer function

For the todo app, this may look like this:

```
const initialState = {
  todos: [],
  visibilityFilter: 'SHOW_ALL'
};
const todosReducer = (state = initialState, action) => {
  // rest of todosReducer logic omitted
};
```

The Recipes application will have the following three slices:

1. `allRecipes`: an array of all recipe objects
2. `favoriteRecipes`: an array of recipe objects chosen by the user from `state.allRecipes`
3. `searchTerm`: a string that filters which recipes are displayed

An example of the store's state may look like this:

```
state = {
  allRecipes: [
    {id: 0, name: 'Jjampong', img: 'img/jjampong.png' },
    {id: 2, name: 'Cheeseburger', img: 'img/cheeseburger.png'
},
    //… more recipes omitted
  ],
  favoriteRecipes: [
    {id: 1, name: 'Doro Wat', img: 'img/doro-wat.png' },
  ],
  searchTerm: 'Doro'
};
```

Notice that each recipe is represented as an object with an `id`, `name`, and `img` property.

Now that you know what the state structure looks like, the first step is to create an `initialState` object.

**Instructions**

**1.**

In the **store.js** file, begin by declaring a new variable called `initialState` and assign to it an empty object.

Hint

Your code should look like this:

```
const initialState = {};
```

**2.**

Now let's add slices to the `initialState` object.

First, add an `allRecipes` property to the `initialState` object with an initial value of an empty array.

This array will be filled once we fetch the data from a database.

**3.**

Next, add a `favoriteRecipes` property to the `initialState` object, also with an initial value of an empty array.

The user will select which recipes to add to this slice as their favorites.

**4.**

Finally, add a `searchTerm` property to the `initialState` object with an initial value of an empty string.

The user will change this value by using a search input field.

Hint

At this point, your `initialState` object should look like this:

```
const initialState = {
  allRecipes: [],
  favoriteRecipes: [],
  searchTerm: ''
};
```

**`store.js`**

```
const initialState = {
  allRecipes: [],
  favoriteRecipes: [],
  searchTerm: ''
};
```

---

### Actions and Payloads For Complex State

The `initialState` structure has been defined and you know that the state of this application has 3 slices: `allRecipes`, `favoriteRecipes`, and `searchTerm`. Now, you can begin thinking about how the user will trigger changes to these slices of state through actions.

*Remember, actions in Redux are represented by plain JavaScript objects that have a `type` property and are dispatched to the store using the `store.dispatch()` method.*

When an application state has multiple slices, individual actions typically only change one slice at a time. Therefore, it is recommended that each action's `type` follows the pattern `'sliceName/actionDescriptor'`, to clarify which slice of state should be updated.

For example, in a todo application with a `state.todos` slice, the action type for adding a new todo might be `'todos/addTodo'`.

For the Recipes application, what do you think some of the action `type` strings might be? What user interactions might trigger them to be dispatched?

Write some of your ideas down before revealing the actions you will be using:

1. `'allRecipes/loadData'`: This action will be dispatched to fetch the needed data from an API right when the application starts.
2. `'favoriteRecipes/addRecipe'`: This action will be dispatched any time the user clicks on the ♡ icon of a recipe from the full set of recipes.
3. `'favoriteRecipes/removeRecipe'`: This action will be dispatched any time the user clicks on the ♡ icon of a recipe from their list of favorites.
4. `'searchTerm/setSearchTerm'`: This action will be dispatched any time the user changes the text of the search input field to filter the full set of recipes.
5. `'searchTerm/clearSearchTerm'`: This action will be dispatched any time the user clicks on the "X" button next to the search input field.

It's also important to consider which of these actions will have a `payload` — additional data passed to the reducer in order to carry out the desired change-of-state. For example, consider the actions for the `searchTerm` slice:

```
store.dispatch({
  type: 'searchTerm/setSearchTerm',
  payload: 'Spaghetti'
});
// The resulting state: { ..., searchTerm: 'Spaghetti' }

store.dispatch({
  type: 'searchTerm/clearSearchTerm'
});
// The resulting state: { ..., searchTerm: '' }
```

- When the learner types in a search term, that data needs to be sent to the store so that the React components know which recipes to render and which to hide.
- When the user clears the search field, no additional data needs to be sent because the store can simply set the search term to be an empty string again.

Once you have a clear idea of the types of actions that will be dispatched in your application, when they will be dispatched, and what `payload` data they will carry, the next step is to make action creators.

*Remember, action creators are functions that return a formatted action object.*

Action creators enable Redux programmers to re-use action object structures without typing them out by hand and they improve the readability of their code, particularly when dealing with bulky `payload`s.

Take a look at **store.js** where you will find that action creators for the two actions above have been defined for you. Your job is to create the remaining three: `loadData()`, `addRecipe()`, and `removeRecipe()`

**Instructions**

**1.**
Open up **./data.js** and you will see an array of recipe objects called `allRecipesData` is exported. Back in **store.js**, at the top of the file, this array is imported (later on, you will fetch data from an API rather than importing from a local file).

This array needs to be sent to the store so that it can populate the `state.allRecipes` slice, which is initially empty. This can be done using the `loadData()` action creator.

Complete the function `loadData()` such that it returns an action object with the following properties:

- `type`: The slice being modified is `state.allRecipes` and the action name is `'loadData'`
- `payload`: The `allRecipesData` array.

*Remember to use the 'sliceName/actionName' pattern for `type`.*

Hint
Action creators typically follow this pattern:

```
const actionName = () {
  return {
    type: 'sliceName/actionName',
    payload: someData
  }
}
```

**2.**

Next up is `addRecipe()` which should be dispatched when the user clicks on the ♡ icon of a particular recipe.

Notice that this function accepts a `recipe` parameter. The `recipe` object then needs to be sent to the store to be added to the `state.favoriteRecipes` slice. For example, this action might be dispatched like so:

```
const exampleRecipe = {
  id: 4,
  name: 'Cheeseburger',
  img: 'img/cheeseburger.jpg'
}
store.dispatch(addRecipe(exampleRecipe));
```

Complete the function called `addRecipe()` such that it returns an action object with the following properties:

- `type`: The slice being modified is `state.favoriteRecipes` and the action name is `'addRecipe'`
- `payload`: The `recipe` object parameter.

---

Hint

Action creators that use parameters often pass them directly to the store as a payload:

```
const actionName = (data) => {
  return {
    type: 'sliceName/actionName',
    payload: data
  }
}
```

**3.**

The last action creator is `removeRecipe()` which should be dispatched when the user clicks on the 💔 icon of a favorited recipe.

`removeRecipe()` also accepts a `recipe` parameter. The `recipe` object needs to be sent to the store so it knows which recipe to remove from the `state.favoriteRecipes` slice.

Complete the function called `removeRecipe()` such that it returns an action object with the following properties:

- `type`: The slice being modified is `state.favoriteRecipes` and the action name is `'removeRecipe'`
- `payload`: The `recipe` object parameter.

---

Hint
Action creators that use parameters often pass them directly to the store as a `payload`:

```
const actionName = (data) => {
  return {
    type: 'sliceName/actionName',
    payload: data
  }
}
```

**store.js**

```
import allRecipesData from './data.js';

const initialState = {
  allRecipes: [],
  favoriteRecipes: [],
  searchTerm: ''
};

// Dispatched when the user types in the search input.
// Sends the search term to the store.
const setSearchTerm = (term) => {
  return {
    type: 'searchTerm/setSearchTerm',
    payload: term
  };
}

// Dispatched when the user presses the clear search button.
```

```
const clearSearchTerm = () => {
  return {
    type: 'searchTerm/clearSearchTerm'
  };
}

// Dispatched when the user first opens the application.
// Sends the allRecipesData array to the store.
const loadData = () => {
  return {
    type: 'allRecipes/loadData',
    payload: allRecipesData
  }
}

// Dispatched when the user clicks on the heart icon of
// a recipe in the "All Recipes" section.
// Sends the recipe object to the store.
const addRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/addRecipe',
    payload: recipe
  }
}

// Dispatched when the user clicks on the broken heart
// icon of a recipe in the "Favorite Recipes" section.
// Sends the recipe object to the store.
const removeRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/removeRecipe',
    payload: recipe
  }
}
```

**data.js**

```
const allRecipesData = [
  { id: 0, name: 'Biscuits', img: 'img/biscuits.jpg'},
  { id: 1, name: 'Bulgogi', img: 'img/bulgogi.jpg'},
  { id: 2, name: 'Calamari', img: 'img/calamari.jpg'},
  { id: 3, name: 'Ceviche', img: 'img/ceviche.jpg'},
  { id: 4, name: 'Cheeseburger', img: 'img/cheeseburger.jpg'},
  { id: 5, name: 'Churrasco', img: 'img/churrasco.jpg'},
  { id: 6, name: 'Dumplings', img: 'img/dumplings.jpg'},
  { id: 7, name: 'Fish & Chips', img: 'img/fishnchips.jpg'},
  { id: 8, name: 'Hummus', img: 'img/hummus.jpg'},
  { id: 9, name: 'Masala Dosa', img: 'img/masaladosa.jpg'},
  { id: 10, name: 'Pad Thai', img: 'img/padthai.jpg'},
];

export default allRecipesData;
```

---

### Immutable Updates & Complex State

Now that you have defined which changes can occur to your application's state, you need a reducer to execute those changes.

*Remember, the `store`'s reducer function is called each time an action is dispatched. It is passed the `action` and the current `state` as arguments and returns the `store`'s next state.*
The second rule of reducers states that when the reducer is updating the `state`, it must make a copy and return the copy rather than directly mutating the incoming `state`. When the state is a mutable data type, like an array or object, this is typically done using the spread operator (`...`).

Below, the `todosReducer` for a todo app demonstrates this in action:

```
const initialState = {
  filter: 'SHOW_INCOMPLETE',
  todos: [
    { id: 0, text: 'learn redux', completed: false },
    { id: 1, text: 'build a redux app', completed: true },
    { id: 2, text: 'do a dance', completed: false },
  ]
};
```

```
const todosReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'filter/setFilter':
      return {
        ...state,
        filter: action.payload
      };
    case 'todos/addTodo':
      return {
        ...state,
        todos: [...state.todos, action.payload]
      } ;
    case 'todos/toggleTodo':
      return {
        ...state,
        todos: state.todos.map(todo => {
          return (todo.id === action.payload.id) ?
            { ...todo, completed: !todo.completed } :
            todo;
        })
      }
    default:
      return state;
  }
};
```

- The `todosReducer` uses the `initialState` as the default `state` value.
- When a `'filter/setFilter'` action is received, it spreads the old `state`'s contents (`...state`) into a new object before updating the `filter` property with the new filter from `action.payload`.
- When a `'todos/addTodo'` action is received, it does the same except this time, since `state.todos` is a mutable array, its contents are also spread into a new array, with the new todo from `action.payload` added to the end.
- When a `'todos/toggleTodo` action is received, it uses the `.map()` method to create a copy of the `state.todos` array. Additionally, the todo being toggled is found using `action.payload.id` and it is spread into a new object and updated.

*It should be clarified that the `state.todos.map()` method only makes a "shallow" copy of the array, meaning the objects inside share the same references as the originals. Therefore, mutations to the objects within the copy will affect the*

*objects within the original. For now, we can make do with this solution, but you will learn how to bypass this issue in a later lesson on the Redux Toolkit.*
Now, let's create a reducer for the Recipes app! In the **store.js** file, after the `initialState` and your action creators, you can see that this function has already been started for you. In the output terminal, you will see the results of `printTests()` which dispatch some actions to the `store`. Your task is to complete it such that it can handle each of the five action creator types that you had created in the last exercise.

### Instructions

**1.**

First up is the `searchTerm/setSearchTerm` action. This action will be dispatched with a `payload` whose value is the `term` to be set as the new value for `state.searchTerm`.

Within the `switch` statement of `recipesReducer()`, fix the `case` that handles the `'searchTerm/setSearchTerm'` action type.

- For this case, the reducer should return a new state object with an updated `searchTerm` slice set to the new term provided by `action.payload`.

If done correctly, the second state printed to the console should show the search term set to `"cheese"`.

Stuck? Get a hint

**2.**

Now, let's fix the `case` for the `favoriteRecipes/addRecipe` action type. This action will be dispatched with a `payload` whose value is the `recipe` object to be added to the `state.favoriteRecipes` array.

- For this action type, the reducer should return a new state object with an updated `favoriteRecipes` slice.
- The new value should be a new array that includes all the previously added values in addition to the new recipe (from `action.payload`) added to the end.

Remember, you must not mutate the incoming `state` object or the original `state.favoriteRecipes` array!

Stuck? Get a hint

**3.**

The final `case` to fix is for the `favoriteRecipes/removeRecipe` action type. This action will be dispatched with a `payload` whose value is the `recipe` object to be removed from the `state.favoriteRecipes` array.

- For this case, the reducer should return a new state object with an updated `favoriteRecipes` slice.
- The `favoriteRecipes` slice should be a new array that includes all the existing values from `state.favoriteRecipes` except for the recipe from `action.payload`.

We recommend that you use the `.filter()` array method and filter out the element whose `'id'` matches the recipe from `action.payload`.

Hint

To remove a value from a slice that is an array without mutating the original state, you can use the `.filter()` method:

```
{
  ...state,
  sliceName: state.sliceName.filter(element => element.id !== elementToRemove.id)
}
```

In this case, the `elementToRemove` is the `action.payload` value.

**`store.js`**

```
import { createStore } from 'redux';
import allRecipesData from './data.js';

const initialState = {
  allRecipes: [],
  favoriteRecipes: [],
  searchTerm: ''
};

const setSearchTerm = (term) => {
  return {
    type: 'searchTerm/setSearchTerm',
    payload: term
  };
```

```
}

const clearSearchTerm = () => {
  return {
    type: 'searchTerm/clearSearchTerm'
  };
};

const loadData = () => {
  return {
    type: 'allRecipes/loadData',
    payload: allRecipesData
  };
};

const addRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/addRecipe',
    payload: recipe
  };
};

const removeRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/removeRecipe',
    payload: recipe
  };
};

/* Complete this reducer */
const recipesReducer = (state = initialState, action) => {
  switch(action.type) {
    case 'allRecipes/loadData':
      return {
        ...state,
        allRecipes: action.payload
      }
    case 'searchTerm/clearSearchTerm':
      return {
        ...state,
```

```
        searchTerm: ''
      }

    case 'searchTerm/setSearchTerm':
      return {
        ...state,
        searchTerm: action.payload
      }

    case 'favoriteRecipes/addRecipe':
      return {
        ...state,
        favoriteRecipes: [...state.favoriteRecipes, action.payload]
      };

    case 'favoriteRecipes/removeRecipe':
      return {
        ...state,
        favoriteRecipes: state.favoriteRecipes.filter(element => element.id !== a
ction.payload.id)
      };

    default:
      return state;
  }
};

const store = createStore(recipesReducer);

/* DO NOT DELETE */
printTests();
function printTests() {
  store.dispatch(loadData());
  console.log('Initial State after loading data');
  console.log(store.getState());
  console.log();
  store.dispatch(addRecipe(allRecipesData[0]));
  store.dispatch(addRecipe(allRecipesData[1]));
  store.dispatch(setSearchTerm('cheese'));
```

```
  console.log("After favoriting Biscuits and Bulgogi and setting the search term to 'cheese'")
  console.log(store.getState());
  console.log();
  store.dispatch(removeRecipe(allRecipesData[1]));
  store.dispatch(clearSearchTerm());
  console.log("After un-favoriting Bulgogi and clearing the search term:")
  console.log(store.getState());
}
```

## data.js

```
import { createStore } from 'redux';
import allRecipesData from './data.js';

const initialState = {
  allRecipes: [],
  favoriteRecipes: [],
  searchTerm: ''
};

const setSearchTerm = (term) => {
  return {
    type: 'searchTerm/setSearchTerm',
    payload: term
  };
}

const clearSearchTerm = () => {
  return {
    type: 'searchTerm/clearSearchTerm'
  };
};

const loadData = () => {
  return {
    type: 'allRecipes/loadData',
    payload: allRecipesData
  };
};
```

```
const addRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/addRecipe',
    payload: recipe
  };
};

const removeRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/removeRecipe',
    payload: recipe
  };
};

/* Complete this reducer */
const recipesReducer = (state = initialState, action) => {
  switch(action.type) {
    case 'allRecipes/loadData':
      return {
        ...state,
        allRecipes: action.payload
      }
    case 'searchTerm/clearSearchTerm':
      return {
        ...state,
        searchTerm: ''
      }

    case 'searchTerm/setSearchTerm':
      return {
        ...state,
        searchTerm: action.payload
      }

    case 'favoriteRecipes/addRecipe':
      return {
        ...state,
        favoriteRecipes: [...state.favoriteRecipes, action.payload]
      };
```

```
    case 'favoriteRecipes/removeRecipe':
      return {
        ...state,
        favoriteRecipes: state.favoriteRecipes.filter(element => element.id !== a
ction.payload.id)
      };

    default:
      return state;
  }
};

const store = createStore(recipesReducer);

/* DO NOT DELETE */
printTests();
function printTests() {
  store.dispatch(loadData());
  console.log('Initial State after loading data');
  console.log(store.getState());
  console.log();
  store.dispatch(addRecipe(allRecipesData[0]));
  store.dispatch(addRecipe(allRecipesData[1]));
  store.dispatch(setSearchTerm('cheese'));
  console.log("After favoriting Biscuits and Bulgogi and setting the search term
to 'cheese'")
  console.log(store.getState());
  console.log();
  store.dispatch(removeRecipe(allRecipesData[1]));
  store.dispatch(clearSearchTerm());
  console.log("After un-favoriting Bulgogi and clearing the search term:")
  console.log(store.getState());
}
```

**console**

```
Initial State after loading data
{ allRecipes:
   [ { id: 0, name: 'Biscuits', img: 'img/biscuits.jpg'
},
```

```
    { id: 1, name: 'Bulgogi', img: 'img/bulgogi.jpg' },
    { id: 2, name: 'Calamari', img: 'img/calamari.jpg'
},
    { id: 3, name: 'Ceviche', img: 'img/ceviche.jpg' }
],
  favoriteRecipes: [],
  searchTerm: '' }

After favoriting Biscuits and Bulgogi and setting the
search term to 'cheese'
{ allRecipes:
   [ { id: 0, name: 'Biscuits', img: 'img/biscuits.jpg'
},
    { id: 1, name: 'Bulgogi', img: 'img/bulgogi.jpg' },
    { id: 2, name: 'Calamari', img: 'img/calamari.jpg'
},
    { id: 3, name: 'Ceviche', img: 'img/ceviche.jpg' }
],
```

---

**Reducer Composition**

In the last exercise, you saw how a single reducer was able to handle the logic for updating every slice of the `store`'s state. Though this approach does work for these relatively small examples, as the application state becomes increasingly more complex, managing it all with a single reducer will become impractical.

The solution is to follow a pattern called *reducer composition*. In this pattern, individual *slice reducers* are responsible for updating only one slice of the application's state, and their results are recombined by a `rootReducer` to form a single state object.

```
// Handles only `state.todos`.
const initialTodos = [
  { id: 0, text: 'learn redux', completed: false },
  { id: 1, text: 'build a redux app', completed: true },
  { id: 2, text: 'do a dance', completed: false },
];
const todosReducer = (todos = initialTodos, action) => {
  switch (action.type) {
    case 'todos/addTodo':
      return [...todos, action.payload]
    case 'todos/toggleTodo':
```

```
      return todos.map(todo => {
        return (todo.id === action.payload.id) ?
          { ...todo, completed: !todo.completed } :
          {...todo};
      });
    default:
      return todos;
  }
};

// Handles only `state.filter`
const initialFilter = 'SHOW_INCOMPLETE',
const filterReducer = (filter = initialFilter, action) => {
  switch (action.type) {
    case 'filter/setFilter':
      return action.payload;
    default:
      return filter;
};

const rootReducer = (state = {}, action) => {
  const nextState = {
    todos: todosReducer(state.todos, action),
    filter: filterReducer(state.filter, action)
  };
  return nextState;
};

const store = createStore(rootReducer);
```

In the reducer composition pattern, when an `action` is dispatched to the `store`:

- The `rootReducer` calls each slice reducer, regardless of the `action.type`, with the incoming `action` and the appropriate slice of the state as arguments.
- The slice reducers each determine if they need to update their slice of state, or simply return their slice of state unchanged.
- The `rootReducer` reassembles the updated slice values in a new state object.

One major advantage of this approach is that each slice reducer only receives its slice of the entire application's state. Therefore, each slice reducer only needs to immutably update its own slice and doesn't care about the others. This removes the problem of copying potentially deeply nested state objects.

Take a look at **store.js** where you will find that the reducer for the Recipe app that you wrote in the last exercise (which can be found in **reducer-old.js**) has been partially rewritten to follow the reducer composition pattern:

- The `initialState` object has been replaced by individual `initialSliceName` variables which are used as default values for each slice reducer's slice of state. This is another common feature of the reducer composition pattern.
- The `allRecipesReducer` and `searchTermReducer` slice reducers have been created for you. Notice that they each have a `default` case.
- Both slice reducers are called within the `rootReducer` to update their respective slices of state.

All that's left is to complete the `favoriteRecipesReducer()` and include it in the `rootReducer()`!

**Instructions**

**1.**
Currently, the default `favoriteRecipes` value for `favoriteRecipesReducer()` is the string `'REPLACE_ME'`. Let's fix that.

First, declare a variable named `initialFavoriteRecipes` and assign it to an empty array (`[]`).

Then, assign the default `favoriteRecipes` value for `favoriteRecipesReducer()` to `initialFavoriteRecipes`.

Hint
To set a default parameter value, use the ES6 syntax:

```
const myFunction = (param = 'default value') => {}
```

**2.**
Next, complete the `favoriteRecipesReducer` such that it immutably updates the `state.favoriteRecipes` slice in response to the following `action.type` cases:

- `'favoriteRecipes/addRecipe'`: Return a new array with all of the prior values of `favoriteRecipes` with the `action.payload` value added to the end.
- `'favoriteRecipes/removeRecipe'`: Return a new array with all of the prior values of `favoriteRecipes` with the `action.payload` value removed.
- `default`: Return `favoriteRecipes` unchanged.

Refer to **reducer-old.js** for the solution code from the last exercise.

Hint

The `'favoriteRecipes/addRecipe'` action can be handled like so:

```
case 'favoriteRecipes/addRecipe':
  return [...favoriteRecipes, action.payload];
```

Remember, `favoriteRecipesReducer()` receives only the `favoriteRecipes` slice of state. Notice how all mention of the overall `state` object is missing from this solution!

The returned value should be a new array with the prior contents of `favoriteRecipes` copied, plus any changes that need to be made.

**3.**

Well done! Now that you have the `favoriteRecipesReducer()` completed, you can use it within the `rootReducer` to update the `state.favoriteRecipes` slice.

Within `rootReducer()`, add a `favoriteRecipes` property to the `nextState` object.

Then, call `favoriteRecipesReducer()`, passing its slice of `state` and the `action` as arguments, and store the result as the value for `nextState.favoriteRecipes`.

Hint

The next state of the `rootReducer()` can be generated like so:

```
const nextState = {
  sliceA: sliceAReducer(state.sliceA, action),
  sliceB: sliceBReducer(state.sliceB, action)
}
```

Notice that each slice reducer receives its slice of `state` and the `action`.

### store.js

```
import { createStore } from 'redux';
import allRecipesData from './data.js';

// Action Creators
////////////////////////////////////

const addRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/addRecipe',
    payload: recipe
  };
```

```
}

const removeRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/removeRecipe',
    payload: recipe
  };
}

const setSearchTerm = (term) => {
  return {
    type: 'searchTerm/setSearchTerm',
    payload: term
  }
}

const clearSearchTerm = () => {
  return {
    type: 'searchTerm/clearSearchTerm'
  };
}

const loadData = () => {
  return {
    type: 'allRecipes/loadData',
    payload: allRecipesData
  };
}

// Reducers
////////////////////////////////////////

const initialAllRecipes = [];
const allRecipesReducer = (allRecipes = initialAllRecipes, action) => {
  switch(action.type) {
    case 'allRecipes/loadData':
      return action.payload
    default:
      return allRecipes;
  }
```

```
}

const initialSearchTerm = '';
const searchTermReducer = (searchTerm = initialSearchTerm, action) => {
  switch(action.type) {
    case 'searchTerm/setSearchTerm':
      return action.payload;
    case 'searchTerm/clearSearchTerm':
      return '';
    default:
      return searchTerm;
  }
}

// Create the initial state for this reducer.
const initialFavoriteRecipes = [];
const favoriteRecipesReducer = (favoriteRecipes = initialFavoriteRecipes, action)
 => {
  switch(action.type) {

    // Add action.type cases here.
    case 'favoriteRecipes/addRecipe':
      return [...favoriteRecipes, action.payload];

    case 'favoriteRecipes/removeRecipe':
      return favoriteRecipes.filter(element => element.id !== action.payload.id)
    default:
      return favoriteRecipes;

    // Don't forget to set the default case!

  }
}


const rootReducer = (state = {}, action) => {
  const nextState = {
    allRecipes: allRecipesReducer(state.allRecipes, action),
    searchTerm: searchTermReducer(state.searchTerm, action),
    // Add in the favoriteRecipes slice using the
```

```
    // favoriteRecipesReducer function.
    favoriteRecipes: favoriteRecipesReducer(state.favoriteRecipes, action),
  }
  return nextState;
}

const store = createStore(rootReducer);
```

## reducer-old.js

```
/*
Notice that, for each recognized action type, the entire
state object must be reconstructed by first copying the
contents of the state using `...state`.
*/
const recipesReducer = (state = initialState, action) => {
  switch(action.type) {

    case 'allRecipes/loadData':
      return {
        ...state,
        allRecipes: action.payload
      }

    case 'searchTerm/clearSearchTerm':
      return {
        ...state,
        searchTerm: ''
      }

    case 'searchTerm/setSearchTerm':
      return {
        ...state,
        searchTerm: action.payload
      };

    case 'favoriteRecipes/addRecipe':
      return {
        ...state,
        favoriteRecipes: [...state.favoriteRecipes, action.payload]
```

```
      };

    case 'favoriteRecipes/removeRecipe':
      return {
        ...state,
        favoriteRecipes: state.favoriteRecipes.filter(element => element.id !== a
ction.payload.id)
      };

    default:
      return state;
  }
};
```

---

**combineReducers**

In the reducer composition pattern, the same steps are taken by the `rootReducer` for each slice reducer:

1. call the slice reducer with its slice of the `state` and the `action` as arguments
2. store the returned slice of state in a new object that is ultimately returned by the `rootReducer()`.

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

// todosReducer and filterReducer omitted

const rootReducer = (state = {}, action) => {
  const nextState = {
    todos: todosReducer(state.todos, action),
    filter: filterReducer(state.filter, action)
  };
  return nextState;
};

const store = createStore(rootReducer);
```

The Redux package helps facilitate this pattern by providing a utility function called `combineReducers()` which handles this boilerplate for us:

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

```
// todosReducer and filterReducer omitted.

const reducers = {
    todos: todosReducer,
    filter: filterReducer
};
const rootReducer = combineReducers(reducers);
const store = createStore(rootReducer);
```

Let's break this code down:

- The `reducers` object contains the slice reducers for the application. The keys of the object correspond to the name of the slice being managed by the reducer value.
- The `combineReducers()` function accepts this `reducers` object and returns a `rootReducer` function.
- The returned `rootReducer` is passed to `createStore()` to create a `store` object.

Just as before, when an action is dispatched to the `store`, the `rootReducer()` is executed which then calls each slice reducer, passing along the `action` and the appropriate slice of `state`.

The last 6 lines of this example can be rewritten inline:

```
const store = createStore(combineReducers({
    todos: todosReducer,
    filter: filterReducer
}));
```

Take a look at **store.js** where you will find the slice reducers that you created in the last exercise. Now, however, the `rootReducer()` is missing. Rather than creating this function by hand, you will use `combineReducers()`.

**Instructions**

**1.**

First, at the top of **store.js**, import `combineReducers` from the `redux` library.

Hint

To import multiple named values from a library, you can write:

```
import { valueA, valueB } from 'library';
// or
import { valueA } from 'library';
import { valueB } from 'library';
```

**2.**

`combineReducers()` accepts an object of reducers as its argument. Let's create one!

At the bottom of **store.js**, create a variable called `reducers`. Assign to it an object with three properties: `allRecipes`, `favoriteRecipes`, `searchTerm`. Each property should be assigned its associated slice reducer.

Hint

Your `reducers` object should look like this:

```
const reducers = {
  sliceA: sliceAReducer, // Right.
  sliceB: sliceBReducer, // Right.
  sliceC: sliceCReducer(), // Wrong.
};
```

Make sure not to call the slice reducers.

**3.**

Now, declare another variable called `rootReducer`. Call `combineReducers()` with the `reducers` object as an argument and assign the returned value to `rootReducer`.

Hint

Your code should look something like this:

```
const rootReducer = combineReducers(reducers);
```

**4.**

Finally, pass the `rootReducer` to the `createStore()` function and save the returned value in a new variable called `store`.

Hint

Your code should look something like this:

```
const store = createStore(rootReducer);
```

**store.js**

```
// Import combineReducers from redux here.
import { createStore, combineReducers } from 'redux';
import allRecipesData from './data.js';

// Action Creators
////////////////////////////////////////

const addRecipe = (recipe) => {
```

```
  return {
    type: 'favoriteRecipes/addRecipe',
    payload: recipe
  };
}

const removeRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/removeRecipe',
    payload: recipe
  };
}

const setSearchTerm = (term) => {
  return {
    type: 'searchTerm/setSearchTerm',
    payload: term
  }
}

const clearSearchTerm = () => {
  return {
    type: 'searchTerm/clearSearchTerm'
  };
}

const loadData = () => {
  return {
    type: 'allRecipes/loadData',
    payload: allRecipeData
  };
}

// Reducers
////////////////////////////////////////

const initialAllRecipes = [];
const allRecipesReducer = (allRecipes = initialAllRecipes, action) => {
  switch(action.type) {
    case 'allRecipes/loadData':
```

```
      return action.payload
    default:
      return allRecipes;
  }
}

const initialSearchTerm = '';
const searchTermReducer = (searchTerm = initialSearchTerm, action) => {
  switch(action.type) {
    case 'searchTerm/setSearchTerm':
      return action.payload
    case 'searchTerm/clearSearchTerm':
      return ''
    default:
      return searchTerm;
  }
}

const initialFavoriteRecipes = [];
const favoriteRecipesReducer = (favoriteRecipes = initialFavoriteRecipes, action)
 => {
  switch(action.type) {
    case 'favoriteRecipes/addRecipe':
      return [...favoriteRecipes, action.payload]
    case 'favoriteRecipes/removeRecipe':
      return favoriteRecipes.filter(recipe => {
        return recipe.id !== action.payload.id
      });
    default:
      return favoriteRecipes;
  }
}

// Create your `rootReducer` here using combineReducers().
const reducers = {
  allRecipes: allRecipesReducer,
  favoriteRecipes: favoriteRecipesReducer,
  searchTerm: searchTermReducer
}
```

```
const rootReducer = combineReducers(reducers);

const store = createStore(rootReducer);
```

### data.js

```
const allRecipesData = [
  { id: 0, name: 'Biscuits', img: 'img/biscuits.jpg'},
  { id: 1, name: 'Bulgogi', img: 'img/bulgogi.jpg'},
  { id: 2, name: 'Calamari', img: 'img/calamari.jpg'},
  { id: 3, name: 'Ceviche', img: 'img/ceviche.jpg'},
  { id: 4, name: 'Cheeseburger', img: 'img/cheeseburger.jpg'},
  { id: 5, name: 'Churrasco', img: 'img/churrasco.jpg'},
  { id: 6, name: 'Dumplings', img: 'img/dumplings.jpg'},
  { id: 7, name: 'Fish & Chips', img: 'img/fishnchips.jpg'},
  { id: 8, name: 'Hummus', img: 'img/hummus.jpg'},
  { id: 9, name: 'Masala Dosa', img: 'img/masaladosa.jpg'},
  { id: 10, name: 'Pad Thai', img: 'img/padthai.jpg'},
];

export default allRecipesData;
```

---

### A File Structure for Redux

At this point, you may have begun thinking that **store.js** is getting pretty long, and yet the Recipes app only has three slices! Imagine trying to fit the logic for an application with a dozen or more slices into one file. That would not be fun.

Instead, it is more common, and a better practice, to break up a Redux application using the Redux Ducks pattern, like so:

```
src/
|-- index.js
|-- app/
    |-- store.js
|-- features/
    |-- featureA/
        |-- featureASlice.js
    |-- featureB/
        |-- featureBSlice.js
```

As you can see in your coding workspace, this file structure has already been set up for you.

All of the Redux logic lives within the top-level directory called **src/**. It contains:

- The entry point for the entire application, **index.js** (we will return to this file in the next exercise).
- The sub-directories **app/** and **features/**.

The **src/app/** directory has only one file (for now), **store.js**. As before, the ultimate purpose of this file is to create the `rootReducer` and the Redux `store`. Now, however, you'll notice that the file is empty! So where did the reducers and action creators go?!

The **src/features/** directory, and its own **src/features/featureX/** sub-directories, contain all of the code relating to each individual slice of the `store`'s state. For example, for the `state.favoriteRecipes` slice, its slice reducer and action creators can be found in the file called **src/features/favoriteRecipes/favoriteRecipesSlice.js**.

Lucky for you, we took care of much of the tedious work involved in refactoring this code. In addition to creating new folders, new files, and copying over the relevant code, this refactor involved exporting each of the slice reducers and action creators, so that they could be imported back into **store.js**.

And that's where you come in!

### Instructions

**1.**

The `reducers` object passed to `combineReducers()` should contain the slice reducers responsible for updating the various slices of the `store`'s state. In the prior lesson, those slice reducers all existed in the same file. Now, you need to import them.

At the top of the **store.js** file, import the following values from their respective files:

- `allRecipesReducer`
- `favoriteRecipesReducer`
- `searchTermReducer`

---

Hint

To import a value from another file you created, you can use the relative path. For example, to import `favoriteRecipesReducer` you can write:

```
import { favoriteRecipesReducer } from
'../features/favoriteRecipes/favoriteRecipesSlice.js'
```

**2.**

Excellent! Now that you have imported the slice reducers, you use them to construct the `reducers` object to be passed to `combineReducers()`.

Within the `reducers` object, add three `key:value` pairs where each `key` is the name of a slice and each `value` is the slice reducer responsible for managing that slice's state.

---

Hint

Your code should look something like this:

```
const reducers = {
  sliceA: sliceAReducer,
  sliceB: sliceBReducer,
  sliceC: sliceCReducer
}
```

**3.**

Now that you have the `reducers` object, you can create the `store` using a combination of the `combineReducers()` and `createStore()` Redux functions.

You are going to do this all in one line of code!

- First call `combineReducers()` with `reducers` as an argument.
- Then, pass the entire `combineReducers(reducers)` function call as an argument to `createStore()`.
- Finally, store the value returned by `createStore()` in a new variable called `store`.

---

Hint

Your code should look something like this:

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

**4.**

Well done! You've reconnected all of the slice reducers from separate files back into the `store` within **src/app/store.js**. In the next exercise, you'll learn

how to build on this application structure by incorporating React components and dispatching actions from them. To do this, the `store` needs to be available to other parts of the application.

Export the `store` value from **src/app/store.js**.

Hint

To export a variable you can write:

```
export const myVariableA = 'some value';
// or
const myVariableB = 'some value';
export myVariableB;
```

### store.js

```
import { createStore, combineReducers } from 'redux';
import { allRecipesReducer } from '../features/allRecipes/allRecipesSlice.js';
import { favoriteRecipesReducer } from  '../features/favoriteRecipes/favoriteRecipesSlice.js';
import { searchTermReducer } from '../features/searchTerm/searchTermSlice.js'


// Import the slice reducers here.

const reducers = {
  // Add the slice properties here
  allRecipes: allRecipesReducer,
  favoriteRecipes: favoriteRecipesReducer,
  searchTerm: searchTermReducer
}

// Declare the store here.
export const store = createStore(combineReducers(reducers));
```

### favoriteRecipeSlice.js

```
export const addRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/addRecipe',
    payload: recipe
```

```
  };
}

export const removeRecipe = (recipe) => {
  return {
    type: 'favoriteRecipes/removeRecipe',
    payload: recipe
  };
}

const initialFavoriteRecipes = [];
export const favoriteRecipesReducer = (favoriteRecipes = initialFavoriteRecipes,
action) => {
  switch(action.type) {
    case 'favoriteRecipes/addRecipe':
      return [...favoriteRecipes, action.payload]
    case 'favoriteRecipes/removeRecipe':
      return favoriteRecipes.filter(recipe => {
        return recipe.id !== action.payload.id
      });
    default:
      return favoriteRecipes;
  }
}
```

---

### Passing Store Data Through the Top-Level React Component

The file structure that you helped implement in the last exercise works nicely when we add in React components. Take a look at the **src** folder in your workspace and you will find the following file structure (new files have a `(+)` next to their name):

```
src/
|-- index.js
|-- app/
    |-- App.js (+)
    |-- store.js
|-- components/
    |-- FavoriteButton.js (+)
```

```
    |-- Recipe.js (+)
|-- features/
    |-- allRecipes/
        |-- AllRecipes.js (+)
        |-- allRecipesSlice.js
    |-- favoriteRecipes/
        |-- FavoriteRecipes.js (+)
        |-- favoriteRecipesSlice.js
    |-- searchTerm/
        |-- SearchTerm.js (+)
        |-- searchTermSlice.js
```

*If you look at the actual file structure in your code editor, you may notice a few unfamiliar files / directories not mentioned in the structure above. The **test/** directory and **index.compiled.js** file are used to test your code on Codecademy. You can ignore them.*

The new components are:

- `<App />`: The top-level component for the entire application.
- `<AllRecipes />`: The component for rendering the recipes loaded from the "database".
- `<FavoriteRecipes />`: The component for rendering the recipes favorited by the user.
- `<SearchTerm />`: The component for rendering the search bar that filters the visible recipes.
- `<Recipe />` and `<FavoriteButton />`: Generic components used by `<AllRecipes />` and `<FavoriteRecipes />`

Aside from the generic components, each feature-related React component file is located in the same directory as the slice file that manages the data rendered by that component. For example, **FavoriteRecipes.js** and **favoriteRecipesSlice.js** are both in the **src/features/favoriteRecipes/** directory.

Open the **src/app/App.js** file where the top-level component, `<App />`, is stored. As in most React applications, this top-level component will render each feature-component and pass any data needed by those components as prop values. In Redux applications, the data passed to each feature-component includes:

1. The slice of the `store`'s state to be rendered. For example, the `state.searchTerm` slice is passed to the `<SearchTerm />` component.

2. The `store.dispatch` method to trigger state changes through user interactions within the component. For example, the `<SearchTerm />` component will need to dispatch `setSearchTerm()` actions.

This distribution of the `store.dispatch` method and the slices of state to all of the feature-components, via the `<App />` component, begins in the **index.js** file. Open up the **src/index.js** file where you will see some standard React code for rendering the top-level `<App />` component. You'll notice that the `store` is missing and the `<App />` component isn't receiving any props!

**Instructions**

**1.**
In order to pass the `store`'s current state and its `dispatch` method to the `<App />` component, the `store` must first be imported into the **index.js** file.

At the top of **index.js**, import the `store` from **store.js**.

Hint
The relative path from **index.js** to **store.js** is:

```
./app/store.js
```

The `./` means "starting from the directory of this file..." where "this file" is **index.js**.

**2.**
Next, get the current state of the `store` and pass it to the `<App />` component as a prop called `state`.

*Note: You won't see anything rendered until the next checkpoint!*

Hint
Your `<App />` rendering code should look like this:

```
<App
  state={the_current_state_of_the_store}
/>,
```

**3.**
The `<App />` component isn't rendering yet because it is expecting to receive a `dispatch` method.

Pass the `store.dispatch` method to the `<App />` component as a prop called `dispatch`.

If done correctly, you should see the `<FavoriteRecipes />` and `<AllRecipes />` components rendered (without data, for now)!

Hint

Make sure NOT TO CALL `store.dispatch` when you pass it!

```
// passing a function...
func={myFunc}

// vs. passing a function CALL...
func={myFunc()}
```

**4.**

Why is the recipe data not being rendered? Remember that the `state.allRecipes` slice begins as an empty array and the data is only loaded AFTER the user opens the page. This data fetch is happening but `render` isn't subscribed to changes to the `store` yet!

At the bottom of **index.js**, use `store.subscribe()` to subscribe the `render` function to the `store` such that each time the `store`'s state changes, the entire `<App />` will be re-rendered.

Hint

To subscribe a method to the changes to the `store`, you can write:

```
store.subscribe(myFunction);
```

### index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

import { App } from './app/App.js';
// Import 'store' here.
import { store } from './app/store.js';


const render = () => {
  // Pass `state` and `dispatch` props to <App />
  ReactDOM.render(
    <App
      state={store.getState()}
      dispatch={store.dispatch}
    />,
    document.getElementById('root')
```

```
  )
}
render();
// Subscribe render to changes to the `store`
store.subscribe(render);
```

## App.js

```
import React from 'react';

import { AllRecipes } from '../features/allRecipes/AllRecipes.js';
import { SearchTerm } from '../features/searchTerm/SearchTerm.js';

export function App(props) {
  const {state, dispatch} = props;

  const visibleAllRecipes = getFilteredRecipes(state.allRecipes, state.searchTerm
);
  const visibleFavoriteRecipes = getFilteredRecipes(state.favoriteRecipes, state.
searchTerm);

// You'll add the <FavoriteRecipes /> component in the next exercise!
  return (
    <main>
      <section>
        <SearchTerm
          searchTerm={state.searchTerm}
          dispatch={dispatch}
        />
      </section>
      <section>
        <h2>Favorite Recipes</h2>

      </section>
      <hr />
      <section>
        <h2>All Recipes</h2>
        <AllRecipes
          allRecipes={visibleAllRecipes}
          dispatch={dispatch}
```

```
        />
      </section>
    </main>
  )
}

/* Utility Helpers */

function getFilteredRecipes(recipes, searchTerm) {
  return recipes.filter(recipe => recipe.name.toLowerCase().includes(searchTerm.t
oLowerCase()));
}
```

---

### Using Store Data Within Feature Components

At the end of the last exercise, you were able to pass the current state of the `store` and its `store.dispatch` method to the top-level component, `<App />`. This allowed the `<App />` component to distribute the `dispatch` method and the slices of the `store`'s state to each feature-component.

So it looks like you're done, right? Not quite. Try adding a favorite recipe and you'll see that it just disappears! Take a closer look at **App.js** and you'll notice that the `<FavoriteRecipes />` component is missing. Then, open up **FavoriteRecipes.js** and you'll see that it is also incomplete. Let's fix that.

Plugging in a feature-component to a Redux application involves the following steps:

- Import the React feature-components into the top-level **App.js** file.
- Render each feature-component and pass along the slice of `state` and the `dispatch` method as props.
- Within each feature-component:
    - Extract the slice of `state` and `dispatch` from `props`.
    - Render the component using data from the slice of state.
    - Import any action creators from the associated slice file.
    - Dispatch actions in response to user inputs within the component.

This process is not different from how you implemented a React + Redux application in the past. Now, however, you must consider that the slices of the `store`'s state and the `dispatch` method must be passed through props.

## Instructions

**1.**
Open up the **App.js** file.

First, import the `FavoriteRecipes` component from the **FavoriteRecipes.js** file.

Hint
The relative path to the **FavoriteRecipes.js** file is:

```
../features/favoriteRecipes/FavoriteRecipes.js
```

**2.**
Now, you can add in the `<FavoriteRecipes />` component to the `<App />` component's structure. Like the other two components, you will need to pass the `dispatch` method to the component as a prop.

The slice data passed to `<FavoriteRecipes />` will need to be filtered first based on the value of `state.searchTerm`. The filtered version of `state.favoriteRecipes` has been created for you and stored in the variable `visibleFavoriteRecipes`.

Within the return statement of the `<App />` component, in the space below the `<h2>Favorite Recipes</h2>` element, add in a `<FavoriteRecipes />` component. You should then pass along the following props:

- `favoriteRecipes`: the `visibleFavoriteRecipes` value
- `dispatch`: the `dispatch` method from the `store`.

If you complete this step correctly, you should see a blank square rendered under the "Favorite Recipes" header.

Hint
Take a look at the `<AllRecipes />` component and see how the slice of `state` is first processed by the `getFilteredRecipes()` function. Also see how the `dispatch` method is passed down. Make sure not to call `dispatch`!.

```
<section>
  <h2>All Recipes</h2>
  <AllRecipes
    allRecipes={visibleAllRecipes}
    dispatch={dispatch}
  />
</section>
```

**3.**

Open up the **FavoriteRecipes.js** file. The job of any presentational component in a Redux app is twofold:

1. Render the data for their associated slice of state.
2. Dispatch actions in response to user interaction within the component.

To do these two things, `<FavoriteRecipes />` was given two props: `favoriteRecipes` and `dispatch`.

At the top of `FavoriteRecipes()`, extract these two values from the `props` parameter.

Hint

You can either use object destructuring syntax...

```
const { propA, propB } = props;
```

...use plain object dot notation...

```
const propA = props.propA;
const propB = props.propB;
```

...or directly pull them out of the parameter list

```
const MyComponent = ({propA, propB}) => {
  // MyComponent code
}
```

**4.**

Now that the `FavoriteRecipes()` component has access to the `favoriteRecipes` slice of state, you can render its data instead of the blank box! Take a look at the `return` statement:

```
return (
  <div className="recipes-container">
    {['REPLACE_ME'].map(createRecipeComponent)}
  </div>
);
```

Replace the entire `['REPLACE_ME']` array with the `favoriteRecipes` prop value.

If done correctly, every recipe object within `favoriteRecipes` will be mapped to a `<Recipe />` component and be rendered (try it out!).

Hint

The return statement should now look like this:

```
return (
  <div className="recipes-container">
    {favoriteRecipes.map(createRecipeComponent)}
  </div>
);
```

**5.**

The `<FavoriteRecipes />` component wants to dispatch an action to the store within `onRemoveRecipeHandler()`, but where are the action creators to help create those actions?

Remember, they have been moved to, and exported from, the **favoriteRecipesSlice.js** file!

At the top of **FavoriteRecipes.js**, import the action creator function, `removeRecipe`.

Stuck? Get a hint

**6.**

Finally, the `removeRecipe()` action creator accepts a `recipe` argument.

Within `onRemoveRecipeHandler()`, which receives a `recipe` parameter, dispatch a `removeRecipe()` action with `recipe` as an argument.

Hint

Because the `dispatch` method has been passed directly to the component, you do not need to write `store.dispatch()`. Instead, you can omit the `store` and just write:

```
dispatch(actionCreator(payload));
```

**App.js**

```
import React from 'react';

import { AllRecipes } from '../features/allRecipes/AllRecipes.js';
import { SearchTerm } from '../features/searchTerm/SearchTerm.js';

import { FavoriteRecipes} from '../features/favoriteRecipes/FavoriteRecipes.js';

// Import the FavoriteRecipes component here.

export function App(props) {
  const {state, dispatch} = props;
```

```
  const visibleAllRecipes = getFilteredRecipes(state.allRecipes, state.searchTerm
);
  const visibleFavoriteRecipes = getFilteredRecipes(state.favoriteRecipes, state.
searchTerm);

  // Render the <FavoriteRecipes /> component.
  // Pass `dispatch` and `favoriteRecipes` props.
  return (
    <main>
      <section>
        <SearchTerm
          searchTerm={state.searchTerm}
          dispatch={dispatch}
        />
      </section>
      <section>
        <h2>Favorite Recipes</h2>
        <FavoriteRecipes
          favoriteRecipes={visibleFavoriteRecipes}
          dispatch={dispatch}
        />
      </section>
      <hr />
      <section>
        <h2>All Recipes</h2>
        <AllRecipes
          allRecipes={visibleAllRecipes}
          dispatch={dispatch}
        />
      </section>
    </main>
  )
}

/* Utility Helpers */

function getFilteredRecipes(recipes, searchTerm) {
  return recipes.filter(recipe => recipe.name.toLowerCase().includes(searchTerm.t
oLowerCase()));
}
```

## FavoriteRecipes.js

```
import React from 'react';
import FavoriteButton from "../../components/FavoriteButton";
import Recipe from "../../components/Recipe";
const unfavoriteIconUrl = 'https://static-assets.codecademy.com/Courses/Learn-Redux/Recipes-App/icons/unfavorite.svg'

// Import removeRecipe from favoriteRecipesSlice.js
import { removeRecipe } from './favoriteRecipesSlice.js';

export const FavoriteRecipes = (props) =>{

  // Extract dispatch and favoriteRecipes from props.
   const {favoriteRecipes, dispatch} = props;

  const onRemoveRecipeHandler = (recipe) => {
    // Dispatch a removeRecipe() action.
    dispatch(removeRecipe(recipe));
  };

  // Map the recipe objects in favoriteRecipes to render <Recipe /> components.
  return (
    <div id='favorite-recipes' className="recipes-container">
      {favoriteRecipes.map(createRecipeComponent)}
    </div>
  );

  // Helper Function
  function createRecipeComponent(recipe) {
    return (
      <Recipe recipe={recipe} key={recipe.id}>
        <FavoriteButton
          onClickHandler={() => onRemoveRecipeHandler(recipe)}
          icon={unfavoriteIconUrl}
        >
          Remove Favorite
        </FavoriteButton>
      </Recipe>
```

```
    )
  }

};
```

---

**Review**

Congratulations! You've learned how to build and organize a React+Redux application with multiple slices of state.

By completing this lesson you now know:.

- The `action.payload` property is used to hold additional data that the reducer might need to carry out a given action. The name `payload` is simply a convention and its value can be anything!
- The spread syntax (`...`) and array methods such as `.map()`, `.slice()`, and `.filter()` can be used to immutably update the state of a complex app.
- *Reducer composition* is a design pattern for managing a Redux store with multiple slices.
- The *root reducer* delegates actions to *slice reducers* that are responsible for updating only their assigned slice of the store's state. The root reducer then reassembles the slices into a new state object.
- `combineReducers()` is a method provided by the `redux` library that accepts a collection of reducer functions and returns a `rootReducer` that implements the reducer composition pattern.
- In a Redux application, slice reducers are often written in separate files. This pattern is known as Redux Ducks.

In the Recipes application you completed in the final exercise, the `store` is passed from the entry point (**index.js**) through the main `<App />` component as a prop. The `<App />` component can then pass the slices of the store's state to its sub-components.

This approach is called "prop drilling" or "prop threading" because the props are "threaded" through the top-level component in order to get them to the presentational components. This isn't ideal considering that the top-level component doesn't make use of those props. In the next lesson, you'll learn how you can use the `react-redux` library to avoid "prop threading" and more tricks for building robust React+Redux applications!

## App.js

```
import React from 'react';

import { AllRecipes } from '../features/allRecipes/AllRecipes.js';
import { SearchTerm } from '../features/searchTerm/SearchTerm.js';
// Import the FavoriteRecipes component here.
import { FavoriteRecipes } from '../features/favoriteRecipes/FavoriteRecipes.js';

export function App(props) {
  const {state, dispatch} = props;

  const visibleAllRecipes = getFilteredRecipes(state.allRecipes, state.searchTerm
);
  const visibleFavoriteRecipes = getFilteredRecipes(state.favoriteRecipes, state.
searchTerm);

  // Render the <FavoriteRecipes /> component.
  // Pass `dispatch` and `favoriteRecipes` props.
  return (
    <main>
      <section>
        <SearchTerm
          searchTerm={state.searchTerm}
          dispatch={dispatch}
        />
      </section>
      <section>
        <h2>Favorite Recipes</h2>
        <FavoriteRecipes
          favoriteRecipes={visibleFavoriteRecipes}
          dispatch={dispatch}
        />
      </section>
      <hr />
      <section>
        <h2>All Recipes</h2>
        <AllRecipes
          allRecipes={visibleAllRecipes}
          dispatch={dispatch}
        />
```

```
      </section>
    </main>
  )
}

/* Utility Helpers */

function getFilteredRecipes(recipes, searchTerm) {
  return recipes.filter(recipe => recipe.name.toLowerCase().includes(searchTerm.t
oLowerCase()));
}
```