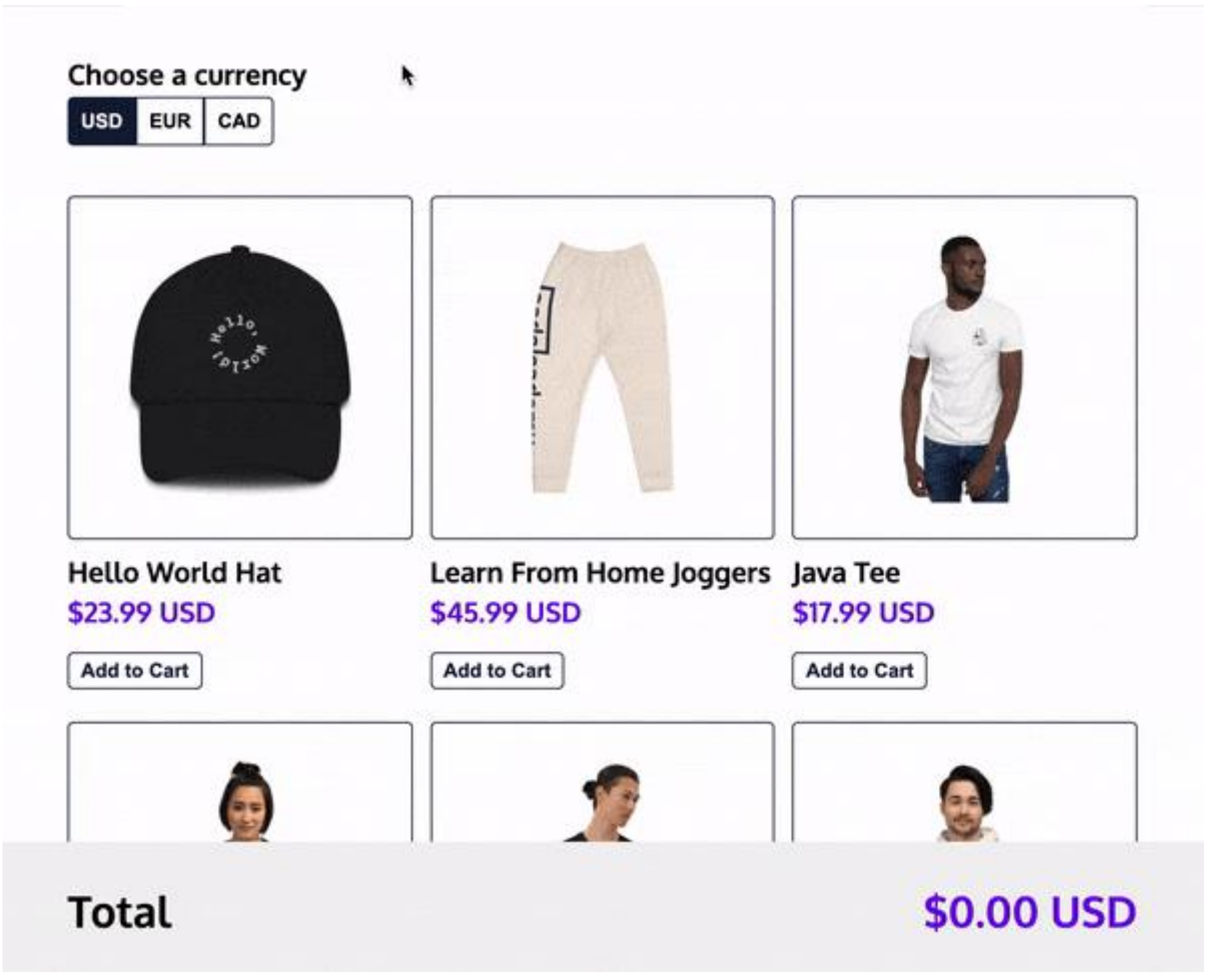
PROJECT CODECADEMY STORE

Codecademy Store

In this project, you'll build a program that mimics Codecademy's own online store! The application should display products from the Codecademy store and allow the user to add them to their cart. In the cart, the user can adjust the quantity of each item and the running total will be displayed at the bottom. Lastly, the user can choose the currency for the entire application.



This application has three slices of state:

- `inventory`: An array of objects representing the items that are available to purchase.
- `cart`: An object that maps the name of each item added to the cart to an object with the price and desired quantity for that item.
- `currencyFilter`: A string that represents the currency used to calculate the prices displayed to the user: `'USD'`, `'CAD'` or `'EUR'`.

An example of this application's state might look like this:

```
state = {
  inventory: [
    { name: 'Hat', img: 'img/hat.png', price: 15.99 },
    { name: 'T-Shirt', img: 'img/t-shirt.png', price: 18.99
},
    { name: 'Hoodie', img: 'img/hoodie.png', price: 49.99 },
  ],
  cart: {
    'Hat': { price: 15.99, quantity: 0 },
    'T-Shirt': { price: 15.99, quantity: 2 },
    'Hoodie': { price: 18.99, quantity: 1 },
  },
  currencyFilter: 'CAD'
}
```

As you will see, the file structure has been organized using the recommended feature-based pattern and most of the inventory and currency features have been built for you. It will be up to you to:

- complete the cart's action creators and reducer logic
- create the `store` using `createStore()` and `combineReducers()`
- pass the `store` resources to presentational components via the top-level `<App />` component
- render the `<Cart />` component using the current state data
- dispatch actions to the `store`

Let's get started!

If you get stuck during this project or would like to see an experienced developer work through it, click **"Get Unstuck"** to **see a project walkthrough video.**

Note: the output terminal below the coding area is there to display syntax errors and can be used when debugging your code. Feel free to minimize it when not in use.

Tasks

16/16 Complete

Mark the tasks as complete by checking them off

Complete the Redux logic for the Cart slice
1.

The first step towards completing the cart feature will be to define the actions that can change the `state.cart` slice, and to handle them in the reducer.

Open up **cartSlice.js** where you will find the `addItem()` action creator as well as the reducer `cartReducer()` which can already handle a `'cart/addItem'` action.

In addition to adding items to the cart, the user should be able to modify the quantity of each item in their cart. First, you will need to create an action creator for this kind of action object.

Below the `addItem()` function:

- Declare a new function called `changeItemQuantity`
- It should have two parameters: `name` (a string) and `newQuantity` (a number)
- It should return an object with two properties: `type` and `payload`
- The `payload` should be an object with a `.name` and `.newQuantity` property.
- Export this function.

Hint

A case for this new action is already written in `cartReducer()`. Make sure your action creator provides the right `type` and `payload` to work with that code.
2.

```
// Example cart state
cart = {
  'Hat': { price: 15.99, quantity: 0 },
  'T-Shirt': { price: 15.99, quantity: 2 },
  'Hoodie': { price: 18.99, quantity: 1 },
},
```

Great! Now that you know what `changeItemQuantity()` actions will look like, you can handle them in the `cartReducer()`. A `case` for this action type has already been started for you. It first pulls out the `name` and `newQuantity` from the payload and grabs the `itemToUpdate` from the `cart`.

The first step is to update this item – but you must do it immutably! Below the variable itemToUpdate…

- Declare a new variable called updatedItem and assign to it a new object.
- Copy the contents of itemToUpdate into updatedItem but set the quantity property to the value of newQuantity.

Hint

You can use the spread operator (...) to copy the contents of one object into another. Then, you can specify any properties that you would like to update:

```
const newObject = {
  ...oldObject,
  prop: newValue
}
```
3.

The final step is to return the new cart state with updatedItem included.
Hint

Take a look at how a new object is added to the cart in the 'cart/addItem' case:

```
return {
  ...cart,
  [name]: newItem
};
```
Notice that the cart's contents are being copied and the name value is used as the key where newItem is added.
Create the Redux store from slice reducers.
4.

With the reducers and action creators ready to go, it's time to set up the store.

Open up **store.js** and, at the top of the file, import the two functions from the 'redux' package used to create the store object: createStore and combineReducers.
Hint

Use the following syntax to import multiple named resources from a module:

```
import { valueA, valueB } from 'module';
```

5.

The `store` needs a `rootReducer` to operate but you currently have three separate slice reducers.

For now, start by importing these reducers into the **store.js** file.

Add three import statements to **store.js**, one for each of the slice reducers:

- `inventoryReducer`
- `cartReducer`
- `currencyFilterReducer`

Hint

You can use relative file paths to import each reducer from their **featureSlice.js** file. For example, to import `inventoryReducer` from **inventorySlice.js** you can write:

```
import { inventoryReducer } from
'../features/inventory/inventorySlice.js'
```

6.

Now that you have imported all of the resources, you can combine the various slice reducers into a `rootReducer` using the `combineReducers` method. Then that `rootReducer` can be used to create the `store` object.

- First, call `combineReducers()` with an object as the argument.
- The object passed to `combineReducers()` should pair each slice name with the appropriate slice reducer
- Next, pass the entire `combineReducers({...})` function call as an argument to `createStore()`.
- Finally, assign the returned value from `createStore()` to a new variable called `store`.
- Make sure you export the `store`!

Hint

Your code should look something like this:

```
export const store = createStore(combineReducers({
  sliceA: sliceAReducer,
  sliceB: sliceBReducer,
  sliceC: sliceCReducer
}));
```
Connect Redux to the top-level React component.
7.

Open up the **index.js** file. This file is known as the “entry point” for the application because it is directly loaded by the **index.html** file and it is responsible for rendering the top-level `<App />` component.

As you can see, the `<App />` component is already being rendered for you, but it is missing the much-needed data from the `store`!

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

In **index.js**:

```
import { store } from './app/store.js';
```
8.

With the `store` imported into **index.js**, you can now pass its data down to the presentational components via the `<App />` component.

The presentational components will need access to the current state of the `store` to render the most up-to-date data. They will also need access to the `store.dispatch` method in order to request new data when the user interacts with the app’s various features.

- Pass the current state of the `store` as a prop called `state` to the `<App />` component
- Pass the `store.dispatch` method as a prop called `dispatch` to the `<App />` component
- Run your program and you should see the currency buttons rendered at the top of the screen and the text “Sorry, no products are currently available…”.

Hint

You can use the `store.getState()` method to get the current state value of the `store`.

When passing the `store.dispatch` method, make sure not to call it!

```
<App
  state={store.getState()}
  dispatch={store.dispatch}
/>
```

9.

The products are not being rendered yet because the product data is only fetched AFTER the page first loads. If you take a look at **src/features/inventory/Inventory.js** you will see that this component dispatches a `loadData()` action upon mounting.

You need to make sure that when any state changes occur, the components are re-rendered with the most up-to-date data.

- At the bottom of **index.js** subscribe the `render` function to changes to the state of the `store`.
- Run your program and you should see the full inventory rendered to the screen!

Hint

You can use the `store.subscribe()` method to subscribe a function to changes to the state of the `store`:

```
store.subscribe(listener);
```

Render the Cart component with data from the store.

10.

Open up **App.js** and you can see that the `<CurrencyFilter />` and `<Inventory />` presentational components are being rendered with their slice of state data and the dispatch method, but the `<Cart />` component is missing!

Let’s add it in.

- At the top of **App.js**, import the `Cart` component from **Cart.js**.

Hint

In **App.js**

```
import { Cart } from  '../features/cart/Cart.js';
```

11.

Now, let’s add the `<Cart />` into the `<App />` component’s structure. Like the other two components, the `<Cart />` will need access to its slice of `state` and the `dispatch` method. It will also need access to the `currencyFilter` slice of `state` to calculate the total cart price.

Inside the `App()` component’s `return` statement…

- Add in the `<Cart />` component below the `<Inventory />` component.
- The `<Cart />` component should have three prop values: `cart`, `currencyFilter`, and `dispatch`.

If done correctly, you should see the cart feature rendered to the screen with a total of `0` and the text `'REPLACE_ME"` in the place of the item list.
Hint

- The `cart` and `currencyFilter` prop values should be the appropriate slice of the `state` prop passed down via `<App />`.
- The `dispatch` prop value should be the `dispatch` prop passed down to `<App />`.

Your code should look something like this:

```
<SliceAComponent
 sliceA={state.sliceA}
 otherSlice={state.otherSlice}
 dispatch={dispatch}
/>
```

Use store data in the Cart component.

12.

Open up **Cart.js** and take a look at the `return` statement. Notice that it is trying to render the variable `cartElements`, which is currently holding the string `'REPLACE_ME'`.

Instead, `cartElements` should be an array of `<li>` elements created using the `createCartItem()` helper function defined at the bottom of the file.

Recall that the `cart` slice of state is an object where each key is the name of an item in the cart. Do the following to make the desired `cartElements` array:

- Initialize `cartElements` to an empty array.
- Iterate through the keys of the `cart` object
- For each key, which is the name of an item, call `createCartItem()` with that item name as an argument.
- Store the values returned by `createCartItem()` in `cartElements`.

```
// Example cart state
cart = {
  'Hat': { price: 15.99, quantity: 0 },
  'T-Shirt': { price: 15.99, quantity: 2 },
  'Hoodie': { price: 18.99, quantity: 1 },
}

// Desired outcome:
cartElements = [
  createCartItem('Hat'),
  createCartItem('T-Shirt'),
  createCartItem('Hoodie'),
]
```

Hint

There are two ways to go about this:

```
const cartElements = [];

for (let itemName in cart) {
  cartElements.push(createCartItem(itemName));
}
```

or…

```
const cartElements = Object.keys(cart).map(createCartItem)
```

13.

Try adding items to your cart. They now show up! However, there are a few things wrong. Most obviously, the cart total is not showing up properly.

At the top of the **Cart.js** file, the `calculateTotal` helper function is imported from the **src/utilities/utilities.js** file. As the name suggests, you can use this function to calculate the cart's total!

- Call `calculateTotal()` with the `cart` and `currencyFilter` prop values as arguments and store the result in the variable `total`.

Hint

Your code should look like this:

```
const total = calculateTotal(cart, currencyFilter);
```

14.

Wonderful! You can now add items to the cart and the total will accurately reflect the cart. However, trying to change the quantity of the items using the number picker doesn't seem to update the state.

Within the `createCartItem()` function, take a look at the `onChange` value. It's using the `onInputChangeHandler()` function, passing along the `name` of the item and the new value of the input field (`e.target.value`).

Now, take a look at `onInputChangeHandler()`. After receiving the `name` and `input` values and doing some data validation and normalization on the `input`, it should dispatch a `changeItemQuantity()` action to the `store` to update the data.

- At the top of **Cart.js**, import the `changeItemQuantity()` action creator that you made earlier in this project.

Hint

cartSlice.js is in the same folder as **Cart.js** so the relative file path is just:

```
'./cartSlice.js`
```

15.

At the end of `onInputChangeHandler()`…

- Use the dispatch method from the props to dispatch a changeItemQuantity() action with name as the first argument and newQuantity as the second.
- After completing this step, try modifying the quantity using the number input field!

Hint

Your code should look like this:

```
dispatch(changeItemQuantity(name, newQuantity));
```

Extra Challenges

16.

Well done! You’ve gone through the entire process of making action creators, setting up a slice reducer, creating the store object, and plugging in the store data into React components. If you’d like to keep working on this project, try implementing this bonus feature:

- Add a search feature (like in the Recipes app) to filter the products shown in the inventory.

Hint

The **src/features/searchTerm/** directory has already been created for you with a completed **SearchTerm.js** component file. It is up to you to

- Complete searchTermSlice.js by creating and exporting the slice reducer and action creators.
- Add the slice reducer to the rootReducer for the store.
- Render the component in the <App /> with the appropriate data.

To filter out the inventory values, you can use this function:

```
function getFilteredItems(items, searchTerm) {
  return items.filter(items =>
items.name.toLowerCase().includes(searchTerm.toLowerCase()))
;
}
```

cartSlice.js

```
export const addItem = (itemToAdd) => {
  return {
    type: 'cart/addItem',
    payload: itemToAdd,
  };
};

// Create your changeItemQuantity action creator here.
export const changeItemQuantity = (name, newQuantity) => {
  return {
    type: 'cart/changeItemQuantity',
    payload: {
      name: name,
      newQuantity: newQuantity
    }
  };
}


const initialCart = {};
export const cartReducer = (cart = initialCart, action) => {
  switch (action.type) {
    case 'cart/addItem': {
      const { name, price } = action.payload;

      // if the item already exists, increase the quantity b
y 1, otherwise set it to 1
      const quantity = cart[name] ? cart[name].quantity + 1
: 1;
      const newItem = { price, quantity };

      // Add the new item to the cart (or replace it if it e
xisted already)
      return {
        ...cart,
        [name]: newItem
      };
    }
    case 'cart/changeItemQuantity': {
      const { name, newQuantity } = action.payload;
      const itemToUpdate = cart[name];

      // Create a copy of itemToUpdate and update the quanti
ty prop.
      const updatedItem = {
        ...itemToUpdate,
        quantity: newQuantity
      }

      // Return a copy of the cart with the updatedItem incl
uded.
      return {
        ...cart,
        [name]: updatedItem
      };
    }
    default: {
      return cart;
    }
  }
};
```

store.js

```
// Import createStore and combineReducers here.
import { createStore, combineReducers } from 'redux';

// Import the slice reducers here.
import { inventoryReducer } from '../features/inventory/inve
ntorySlice.js';

import { cartReducer } from '../features/cart/cartSlice.js';

import { currencyFilterReducer } from '../features/currencyF
ilter/currencyFilterSlice.js';
// Create and export the store here.
import { searchTermReducer } from '../features/searchTerm/se
archTermSlice.js';

export const store = createStore(combineReducers({
     cart: cartReducer,
     inventory: inventoryReducer,
     currencyFilter: currencyFilterReducer,
     searchTerm: searchTermReducer
}));
```

index.js

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

import { App } from './app/App.js';
// Import the store here.
import { store } from './app/store.js';
// Pass state and dispatch props to the <App /> component.
const render = () => {
  ReactDOM.render(
    <App
      state={store.getState()}
      dispatch={store.dispatch}
    />,
    document.getElementById('root')
  )
};
render();

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

App.js

```
import React from 'react';

import { Inventory } from '../features/inventory/Inventory.j
s';
import { CurrencyFilter } from '../features/currencyFilter/C
urrencyFilter.js';
// Import the Cart component here.
import { Cart } from '../features/cart/Cart.js';
import { SearchTerm } from '../features/searchTerm/SearchTerm.js'

// Render the Cart component below <Inventory />
export const App = (props) => {

  const { state, dispatch } = props;

  return (
    <div>
      <CurrencyFilter
        currencyFilter={state.currencyFilter}
        dispatch={dispatch}
      />

      <SearchTerm
        searchTerm={state.searchTerm}
        dispatch={dispatch}
      />

      <Inventory
        inventory={getFilteredItems(state.inventory, state.searchTerm)}
        currencyFilter={state.currencyFilter}
        dispatch={dispatch}
      />

      <Cart
        cart={state.cart}
        currencyFilter={state.currencyFilter}
        dispatch={dispatch}
      />

    </div>
  );
};

function getFilteredItems(items, searchTerm) {
  return items.filter(items => items.name.toLowerCase().includes(searchTerm.toLowerCase()));
}
```

Cart.js

```
import React from 'react';
import {
  calculateTotal,
  getCurrencySymbol,
} from '../../utilities/utilities.js';

// Import the changeItemQuantity() action creator.
import { changeItemQuantity } from './cartSlice.js';

export const Cart = (props) => {
  const { cart, currencyFilter, dispatch } = props;

  const onInputChangeHandler = (name, input) => {
    // If the user enters a bad value...
    if (input === '') {
      return;
    }

    // Otherwise, convert the input into a number and pass i
t along as the newQuantity.
    const newQuantity = Number(input);

    // Dispatch an action to change the quantity of the give
n name and quantity.
    dispatch(changeItemQuantity(name, newQuantity));

  };

  // Use the cart and currencyFilter slices to render their
data.
  const cartElements = [];
  for (let itemName in cart) {
  cartElements.push(createCartItem(itemName));
}
  const total = calculateTotal(cart, currencyFilter);
  return (
    <div id="cart-container">
      <ul id="cart-items">{cartElements}</ul>
      <h3 className="total">
        Total{' '}
        <span className="total-value">
          {getCurrencySymbol(currencyFilter)}{total} {curren
cyFilter}
        </span>
      </h3>
    </div>
  );

  function createCartItem(name) {
    const item = cart[name];

    if (item.quantity === 0) {
      return;
    }

    return (
      <li key={name}>
        <p>{name}</p>
        <select
          className="item-quantity"
          value={item.quantity}
          onChange={(e) => {
            onInputChangeHandler(name, e.target.value);
          }}
        >
          {[...Array(100).keys()].map((_, index) => (
            <option key={index} value={index}>
              {index}
            </option>
          ))}
        </select>
      </li>
    );
  }
};
```