实验：使用 React Toolkit 渲染电子商务数据

预计所需时间: 60 分钟

介绍

在这个实验中，您将学习如何使用 Redux Toolkit 管理整个应用程序的状态，以便任何组件都可以访问，而无需遵循组件之间的层次结构。您将使用 React 和 Redux 构建一个简单的电子商务应用程序，在其中显示一系列产品，每个产品都有一个“添加到购物车”按钮，显示添加到购物车的商品，并允许用户从购物车中移除商品。所有这些信息都将通过 Redux Toolkit 在整个应用程序中全局可用。

学习目标

完成此实验后，您将能够：

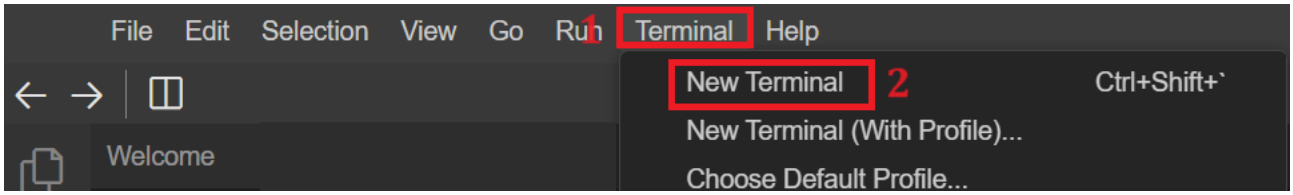
- 将 React 组件与 Redux 集成以进行状态管理
- 实现基本的电子商务功能，例如将商品添加到购物车、从购物车中移除商品以及清空购物车
- 练习组合多个 React 组件以构建一个统一的用户界面

先决条件

- 对 React 组件组合的基本知识
- 中级 JavaScript 知识
- 知道 React 功能组件、使用 useState 钩子的状态管理和 Redux Toolkit

步骤 1：设置环境

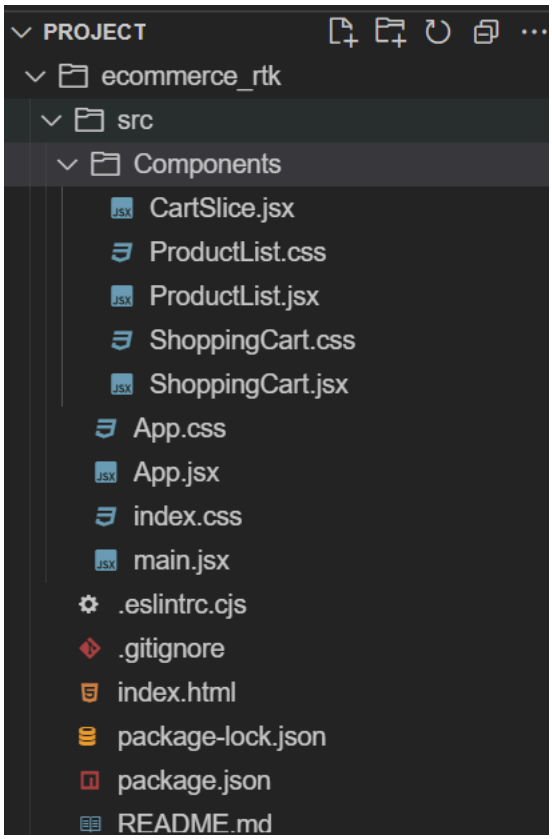
1. 在实验室顶部菜单中，点击窗口右上角标记为 1 的 **Terminal** 选项卡，然后点击标记为 2 的 **New Terminal**。



2. 现在，在终端中输入以下命令以克隆该 React 应用程序的基础模板，然后按 Enter。

```
git clone https://github.com/ibm-developer-skills-network/ecommerce_rtk.git
```

3. 上述命令将在“Project”文件夹下创建一个名为“ecommerce_rtk”的文件夹。您可以在截图中看到结构。



4. 接下来，您需要确保终端的路径已切换到克隆的文件夹，以便对这个 React 应用程序执行某些操作。使用以下命令在终端中导航到 **ecommerce_rtk** 文件夹。

   ```
   cd ecommerce_rtk
   ```

5. 为确保您克隆的代码正常工作，您需要执行以下步骤：

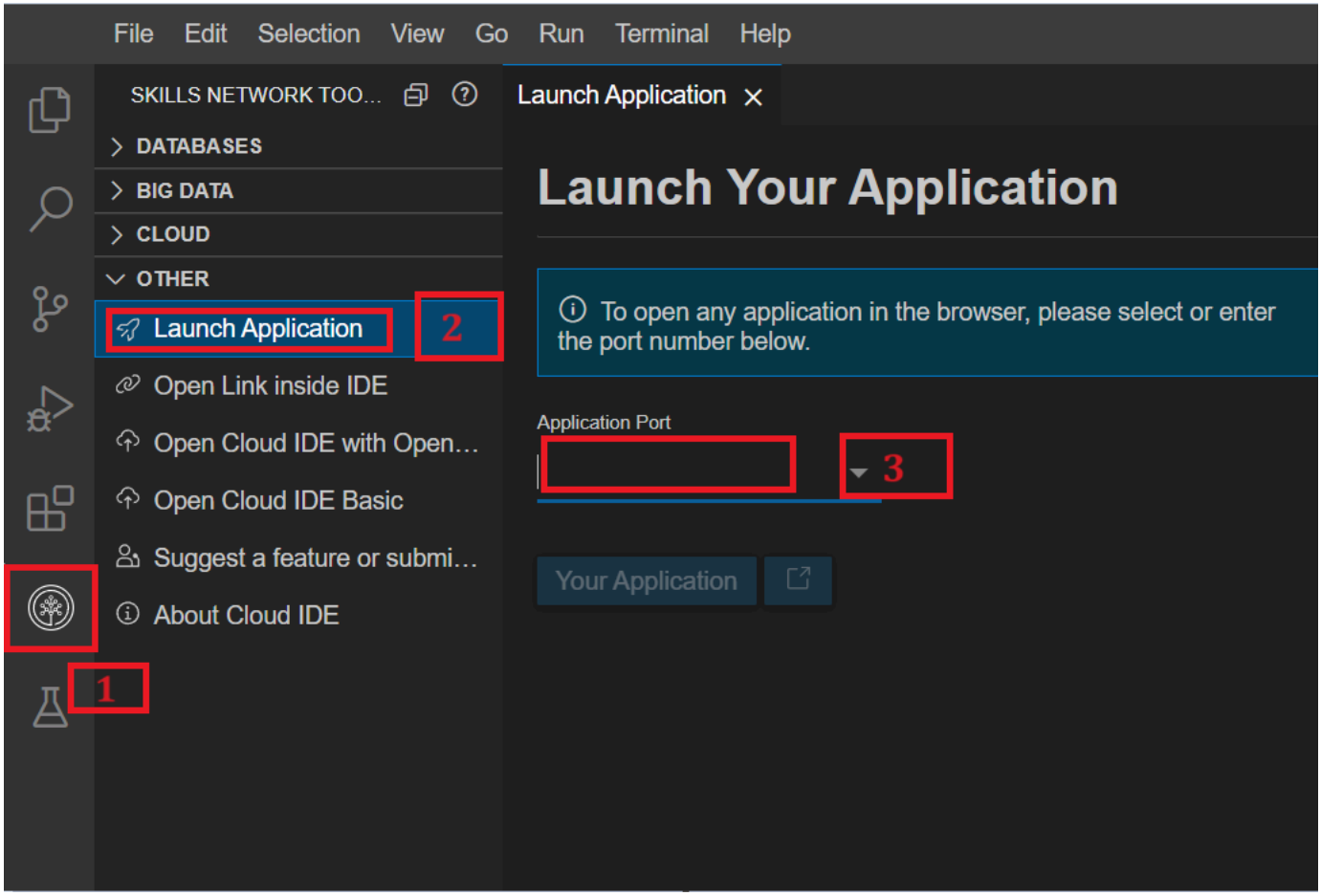
   - 在终端中输入给定命令并按 Enter。这条命令将安装执行应用程序所需的所有必要包。

     ```
     npm install
     ```

   - 然后执行以下命令以运行应用程序，这将为您提供端口号 4173。

     ```
     npm run preview
     ```

6. 要查看您的 React 应用程序，请点击左侧的技能网络图标（参见数字 1）。此操作将打开 **SKILLS NETWORK TOOLBOX**。接下来，点击 **Launch Application**（参见数字 2）。在 **Application Port** 中输入端口号 **4173**（参见数字 3），然后点击 。



7. 输出将显示在给定的截图中。



8. 您可以通过添加、提交并推送到您的 GitHub 存储库来保留您在此实验室的最新工作。这确保即使您没有持续进行任务，您的进度也会被保存，允许您从中断的地方恢复。

注意：第 8 步是可选的。我们建议您一次性完成任务

步骤 2：实现 ProductList 组件

1. 接下来，导航到您克隆的 **ecommerce_rtk** 文件夹中的 **src** 目录下的 **Components** 文件夹中的 **ProductList.jsx** 文件。
2. 该组件的基本结构将如截图所示。

```jsx
import React from 'react';
import './ProductList.css';

const ProductList = () => {

  const products = [
    { id: 1, name: 'Product A', price: 60 },
    { id: 2, name: 'Product B', price: 75 },
    { id: 3, name: 'Product C', price: 30 },
  ];

  return (
    <div className="product-list">
      <h2 className="product-list-title">Products</h2>
      <ul className="product-list-items">

      </ul>
    </div>
  );
};

export default ProductList;
```

3. 现在您需要在前端显示产品列表。为此，您需要在类名为 "product-list-items" 的 `<ul>` 标签内应用 map 方法。

```
{products.map(product => (
  <li key={product.id} className="product-list-item">
  <span>{product.name} - ${product.price}</span>
  <button>
    Add to Cart
  </button>
  </li>
))}
```

4. 检查输出，结果将如给定的截图所示。



步骤 3：实现 Redux 逻辑

1. 你需要应用 Redux 工具包的逻辑，以确保当你点击添加产品按钮将产品添加到购物车时，你输入的产品数量信息应该在任何组件中全局可用。
2. 现在导航到位于你克隆的 **ecommerce_rtk** 文件夹的 **src** 目录中的 **Components** 文件夹中的 **CartSlice.jsx** 文件。
3. 你将看到如下结构：

```
const CartSlice = ({

});
```

4. 首先，在 **CartSlice** 外部初始化一个名为 **cartItems** 的空数组。

```
const initialState = {
  cartItems: [],
};
```

5. 现在使用一个 **createSlice** Redux Toolkit 函数初始化 **CartSlice**。你需要安装 `@reduxjs/toolkit` 和 `react-redux` 作为第三方模块。对于这个实验，你不需要单独安装，因为它已经在 `package.json` 文件中提供，**createSlice** 是 Redux Toolkit 提供的一个工具函数，这是一个建立在 Redux 之上的库。它简化了创建 Redux 切片的过程，切片是 Redux 状态的部分，以及相关的动作创建器和 reducers。

   ```
   const CartSlice = createSlice({

   });
   ```

6. 确保在这个组件的顶部导入 **createSlice**。

   ```
   import { createSlice } from '@reduxjs/toolkit';
   ```

步骤 4：动作和 reducer 的创建

1. 切片创建：
   - 您调用 **createSlice**，并传入一个包含切片配置选项的对象。
   - 配置选项包括：
     - name：一个字符串值，表示切片的名称。它在 Redux Toolkit 内部用于动作类型前缀和其他目的。
     - initialState：一个表示切片初始状态的对象。
     - reducers：一个包含 reducer 函数的对象。每个键值对表示一个单独的 reducer，其中键是动作的名称，值是 reducer 函数。

       ```
       const CartSlice = createSlice({
         name: 'cart',
         initialState,
         reducers: {
         }
       });
       ```

步骤 5：创建 reducers 和导出 actions

1. 在 **reducer** 对象内，您需要创建五个函数，其中两个用于处理购物车中产品的添加和移除，一个用于一次性清空所有物品，另外两个用于增加和减少数量。
   - `addItemToCart`:
     - 此 reducer 函数处理将物品添加到购物车的操作。
     - 它接受两个参数：**state**（切片的当前状态）和 **action**（包含有效载荷的调度操作）。
     - 它首先通过在 **state.cartItems** 中搜索物品的 ID 来检查该物品是否已经存在于购物车中。
     - 如果物品存在（**existingItem** 为真），则将购物车中现有物品的数量增加 1。
     - 如果购物车中不存在该物品，则将物品以数量 1 添加到 **cartItems** 数组中。

       ```
       addItemToCart(state, action) {
             const existingItem = state.cartItems.find(item => item.id === action.payload.id);
             if (existingItem) {
               existingItem.quantity += 1;
             } else {
               state.cartItems.push({ ...action.payload, quantity: 1 });
             }
           },
       ```

   - `removeItemFromCart`:
     - 此 reducer 函数处理从购物车中移除物品的操作。
     - 它接受两个参数：**state** 和 **action**。
     - 它通过过滤掉 **action payload** 中提供的 ID 的物品来更新 **cartItems** 数组。

       ```
       removeItemFromCart(state, action) {
             state.cartItems = state.cartItems.filter(item => item.id !== action.payload);
           },
       ```

   - `clearCart`:
     - 此 reducer 函数处理清空整个购物车的操作。
     - 它只接受状态参数。
     - 它将 **cartItems** 数组设置为空数组，从而有效地清空购物车中的所有物品。

       ```
       clearCart(state) {
             state.cartItems = [];
           },
       ```

   - `increaseItemQuantity`:
     - 此 reducer 函数处理增加购物车中特定物品数量的操作。
     - 它接受两个参数：**state** 和 **action**。
       - `state`: 这表示 Redux 存储的当前状态。它通常包含与应用程序相关的数据。
       - `action`: 这是一个描述发生的操作的对象。Redux 操作是普通的 JavaScript 对象，必须具有一个 type 属性，指示正在执行的操作类型。此外，它们可能包含执行操作所需的其他数据。在这种情况下，action.payload 可能包含需要增加数量的物品的标识符（id）。
     - 函数逻辑：

        - 它在购物车中查找标识符（id）与 action payload 中传递的标识符匹配的物品。
        - 如果找到该物品（itemToIncrease 不为 undefined），则将该物品的数量属性增加 1。

```
increaseItemQuantity(state, action) {
      const itemToIncrease = state.cartItems.find(item => item.id === action.payload);
      if (itemToIncrease) {
        itemToIncrease.quantity += 1;
      }
    },
```

    - `decreaseItemQuantity`:
        - 此 reducer 函数处理减少购物车中特定物品数量的操作。
        - 它接受两个参数：**state** 和 **action**。
            - state: 这表示 Redux 存储的当前状态，通常包含与应用程序相关的所有数据。
            - action: 与前一个函数类似，它是一个描述正在执行的操作的对象。预计它具有一个 type 属性，指示操作类型，并可能包括执行操作所需的其他数据。在这里，action.payload 可能包含需要减少数量的物品的标识符（id）。
        - 函数逻辑：
            - 它尝试在购物车中查找标识符（id）与 action payload 中提供的标识符匹配的物品。
            - 如果找到该物品（itemToDecrease 不为 undefined）并且其数量大于 1，则将该物品的数量属性减少 1。

```
decreaseItemQuantity(state, action) {
      const itemToDecrease = state.cartItems.find(item => item.id === action.payload);
      if (itemToDecrease && itemToDecrease.quantity > 1) {
        itemToDecrease.quantity -= 1;
      }
    },
```

2. 导出 Action Creators 和 Reducer：
    - **createSlice** 返回一个包含生成的 action creators 和 reducer 函数的对象。
    - 然后您可以导出这些 action creators 和 reducer 函数，以便在 Redux 存储设置和整个应用程序中使用。

```
export const {
  addItemToCart,
  removeItemFromCart,
  clearCart,
  increaseItemQuantity,
  decreaseItemQuantity,
} = CartSlice.actions;
export default CartSlice.reducer;
```

您可以将上述代码包含在组件 **CartSlice** 的最后。

第6步：创建 store.js 文件

1. 接下来，创建一个 **store.js** 文件。
2. 选择 **src** 文件夹，右键点击该文件夹。然后选择 **新建文件**，并将名称写为 **store.js**。
3. 在此文件中执行以下操作：
    - 导入 **configureStore** 和 **Reducer**：
        - 代码从 **@reduxjs/toolkit** 导入 **configureStore** 函数，用于创建 Redux 存储。
        - 还从 **CartSlice** 文件中导入了 reducer 函数 **cartReducer**，假设您在该文件中定义了管理购物车状态的切片。
    - 存储配置：
        - **configureStore** 被调用，传入一个包含存储配置选项的对象。
        - reducer 属性被指定为一个对象，其中每个键表示一个状态切片，每个值表示相应的 reducer 函数。
        - 在这种情况下，**cartReducer** 与购物车状态切片相关联。这意味着由 **cartReducer** 管理的状态将存储在 Redux 存储中的 cart 键下。
    - 其他存储配置选项：
        - 可以在传递给 **configureStore** 的对象中包含其他存储配置选项。
        - 例如，您可以包含中间件、增强器或其他选项，例如开发工具配置。
    - 导出存储：
        - 最后，配置好的 Redux 存储（store）从模块中导出，使其在整个应用程序中可用。

```
import { configureStore } from '@reduxjs/toolkit';
import cartReducer from './Components/CartSlice';
const store = configureStore({
  reducer: {
    cart: cartReducer,
  },
});
export default store;
```

4. 现在，为了使这个数据在应用程序中的任何组件中全局可用，您需要在 **main.jsx** 组件中导入数据。为此，请导航到 **main.jsx** 文件，并在文件中粘贴以下代码。

```
import React from 'react'
```

```
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'
import { Provider } from 'react-redux'
import store from './store.js'
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
 <Provider store={store}>
    <App />
    </Provider>
  </React.StrictMode>,
)
```

- 在上面的代码中，**store.js** 文件在 `<React.StrictMode>` 中被导入。来自 react-redux 的 `<Provider>` 通过将 store 作为 props 传递，向其层次结构中的所有组件提供 Redux 存储。这允许组件，包括 `<App />`，访问和与 Redux 存储进行交互以进行状态管理。

步骤 7：全局添加产品和商店数据

1. 在 **ProductList** 组件中，现在初始化两个变量，一个名为 **disabledProducts**，使用 **useState** 钩子，另一个名为 **dispatch**，将利用 **useDispatch()** 的功能。

```
const dispatch = useDispatch();
const [disabledProducts, setDisabledProducts] = useState([]); // 用于存储禁用产品的状态
```

- 确保在组件顶部包含给定的语句。

```
import { useDispatch } from 'react-redux';
import { useState } from 'react';
import { addItemToCart } from './CartSlice';
```

在上述代码中，**addItemToCart** 用于获取 reducer 函数的详细信息，以调度哪个产品被添加到购物车中到 **store.js**。

2. 在 **ProductList** 组件中，实现将数据添加到购物车的功能，并使用 **useDispatch** 钩子直接将数据发送到 `store.js`。在 `<button>` 标签中为 **onClick** 事件调用一个函数 **handleAddToCart**。

```
<button
    className={`add-to-cart-btn ${disabledProducts.includes(product.id) ? 'disabled' : ''}`}
    onClick={() => handleAddToCart(product)}
    disabled={disabledProducts.includes(product.id)} // Disable button if product is in disabledProducts >
    Add to Cart
</button>
```

- 当点击此按钮时，会调用 `handleAddToCart` 函数，并将产品作为参数传递。
  - 按钮的外观是动态决定的，取决于产品是否包含在 `disabledProducts` 数组中，如果产品在数组中或已添加，则禁用按钮。
  - 此功能防止将重复项目添加到购物车，并在必要时通过将按钮样式设置为禁用来提供视觉反馈。

3. 初始化名为 **handleAddToCart** 的函数。

```
const handleAddToCart = product => {
    dispatch(addItemToCart(product));
    setDisabledProducts([...disabledProducts, product.id]); // Mark the product as disabled
  };
```

▼ 点击这里查看 ProductList.jsx 代码

```
import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { addItemToCart } from './CartSlice';
import './ProductList.css'; // 导入组件特定样式的 CSS 文件
const ProductList = () => {
  const dispatch = useDispatch();
  const [disabledProducts, setDisabledProducts] = useState([]); // 存储禁用产品的状态
  const products = [
    { id: 1, name: '产品 A', price: 60 },
    { id: 2, name: '产品 B', price: 75 },
    { id: 3, name: '产品 C', price: 30 },
  ];
  const handleAddToCart = product => {
    dispatch(addItemToCart(product));
    setDisabledProducts([...disabledProducts, product.id]); // 将产品标记为禁用
  };
  return (
    <div className="product-list">
      <h2 className="product-list-title">产品</h2>
      <ul className="product-list-items">
        {products.map(product => (
          <li key={product.id} className="product-list-item">
            <span>{product.name} - ${product.price}</span>
            <button
              className={`add-to-cart-btn ${disabledProducts.includes(product.id) ? 'disabled' : ''}`}
              onClick={() => handleAddToCart(product)}
              disabled={disabledProducts.includes(product.id)} // 如果产品在禁用产品中则禁用按钮
            >
              添加到购物车
            </button>
          </li>
        ))}
      </ul>
    </div>
  );
};
export default ProductList;
```

第8步：在购物车中显示产品

1. 现在在 **ShoppingCart.jsx** 组件中，您将创建逻辑以显示用户添加到购物车中的商品。它使用一种叫做 Redux 的特殊工具来管理购物车，并跟踪用户正在购买的内容。该组件让用户查看购物车中的商品、价格以及每种商品的数量。用户还可以从购物车中删除商品或更改每种商品的数量。这就像拥有一个虚拟购物篮，帮助用户跟踪他们将要购买的商品。
2. 为了实现这一点，请导航到 src 文件夹下的 **ShoppingCart.jsx** 组件。
3. 该组件的基本结构如下所示。

```jsx
import React from 'react';
import './ShoppingCart.css';

const ShoppingCart = () => {

  return (
    <>
    <div className="shopping-cart">
      <h2 className="shopping-cart-title">Shopping Cart</h2>
      <ul className="cart-items">

      </ul>
      <button className="clear-cart-btn">Clear Cart</button>
    </div>

    </>
  );
};

export default ShoppingCart;
```

4. 实现给定的功能：–
   - *导入*：该组件导入必要的依赖项：React、useDispatch、useSelector 来自 react-redux，以及来自 CartSlice 的动作创建器（removeItemFromCart、clearCart、increaseItemQuantity、decreaseItemQuantity）。

     ```
     import { useDispatch, useSelector } from 'react-redux';
     import { removeItemFromCart, clearCart, increaseItemQuantity, decreaseItemQuantity } from './CartSlice'; // 假设您有增加和减
     import './ShoppingCart.css';
     ```

   - *函数组件*：ShoppingCart 组件是一个使用箭头函数语法声明的函数组件。
   - *Redux Hooks*：该组件使用来自 react-redux 的 **useDispatch** 和 **useSelector** 钩子与 Redux 存储进行交互。**useDispatch** 用于分发动作，**useSelector** 用于从 Redux 存储中提取数据。
   - *状态检索*：**cartItems** 变量通过选择 **state.cart.cartItems** 从 Redux 存储的状态中检索商品数组。**totalAmount** 通过迭代 **cartItems** 并将每个商品的价格乘以其数量，然后求和来计算总金额。

     ```
     const dispatch = useDispatch();
       const cartItems = useSelector(state => state.cart.cartItems);
       const totalAmount = cartItems.reduce((total, item) => total + item.price * item.quantity, 0);
     ```

     在函数组件的返回之前包含上述代码。
   - *事件处理程序*：**handleRemoveItem** 分发 **removeItemFromCart** 动作，传入要删除的商品的 ID。**handleClearCart** 分发 **clearCart** 动作以清空购物车中的所有商品。**handleIncreaseQuantity** 分发 **increaseItemQuantity** 动作以增加特定商品的数量。**handleDecreaseQuantity** 分发 **decreaseItemQuantity** 动作以减少特定商品的数量。

     ```
     const handleRemoveItem = itemId => {
         dispatch(removeItemFromCart(itemId));
       };
       const handleClearCart = () => {
         dispatch(clearCart());
       };
       const handleIncreaseQuantity = itemId => {
         dispatch(increaseItemQuantity(itemId));
     ```

```
  };
  const handleDecreaseQuantity = itemId => {
    dispatch(decreaseItemQuantity(itemId));
  };
```

- *渲染*：该组件渲染一个购物车用户界面，在 `<ul>` 标签中列出购物车中的每个商品，类名为 **cart-items**，显示商品名称、价格、数量控制（增加或减少数量的按钮）和每个商品的删除按钮。如果总金额大于 0，则在购物车下方显示总金额。

```
{cartItems.map(item => (
            <li key={item.id} className="cart-item">
              <span>{item.name} - ${item.price}</span>
              <div className="quantity-controls">
                <button className="quantity-control-btn" onClick={() => handleDecreaseQuantity(item.id)}>-</button>
                <span> {item.quantity}</span>
                <button className="quantity-control-btn" onClick={() => handleIncreaseQuantity(item.id)}>+</button>
              </div>
              <button className="remove-item-btn" onClick={() => handleRemoveItem(item.id)}>Remove</button>
            </li>
          ))}
```

- *按钮控制*：提供数量控制（- 和 + 按钮）以减少或增加每种商品的数量。单击 - 按钮调用 **handleDecreaseQuantity**，并传入商品的 ID。单击 + 按钮调用 **handleIncreaseQuantity**，并传入商品的 ID。
- 提供一个标记为 **Clear Cart** 的按钮，当单击时会删除购物车中的所有商品。它会触发 handleClearCart 函数。

```
<button className="clear-cart-btn" onClick={handleClearCart}>Clear Cart</button>
```

- 只有在添加了商品的情况下才显示 totalAmount，否则不渲染任何内容。

```
<div>{totalAmount ? <div>'总金额为 {totalAmount}</div> : ''}</div>
```

- 让我们逐步分析：
  - 最外层的 div 元素包含一个花括号 {} 内的表达式。
  - 在表达式内部，有一个三元运算符（条件 ? 表达式1 : 表达式2）用于在 JSX 中进行条件渲染。
  - 如果 totalAmount 为真，则渲染一个嵌套的 div 元素。这个嵌套的 div 包含字符串 '总金额为 {totalAmount}'，其中 {totalAmount} 是意图将 totalAmount 变量的值插入到字符串中的。
  - 如果 totalAmount 为假，则渲染一个空字符串。
  - 三元运算的结果被渲染在最外层的 div 元素内。

单击下面查看 **CartSlice.jsx** 的代码。

▼ 点击这里查看 CartSlice.jsx 的代码

```
import { createSlice } from '@reduxjs/toolkit';
const initialState = {
    cartItems: [],
  };
const CartSlice = createSlice({
    name: 'cart',
    initialState,
    reducers: {
        addItemToCart(state, action) {
            const existingItem = state.cartItems.find(item => item.id === action.payload.id);
            if (existingItem) {
              existingItem.quantity += 1;
            } else {
              state.cartItems.push({ ...action.payload, quantity: 1 });
            }
          },
          removeItemFromCart(state, action) {
            state.cartItems = state.cartItems.filter(item => item.id !== action.payload);
          },
          clearCart(state) {
            state.cartItems = [];
          },
          increaseItemQuantity(state, action) {
            const itemToIncrease = state.cartItems.find(item => item.id === action.payload);
            if (itemToIncrease) {
              itemToIncrease.quantity += 1;
            }
          },
          decreaseItemQuantity(state, action) {
            const itemToDecrease = state.cartItems.find(item => item.id === action.payload);
            if (itemToDecrease && itemToDecrease.quantity > 1) {
              itemToDecrease.quantity -= 1;
            }
          },
    }
});
export const {
    addItemToCart,
    removeItemFromCart,
    clearCart,
    increaseItemQuantity,
    decreaseItemQuantity,
  } = CartSlice.actions;
  export default CartSlice.reducer;
```

单击下面查看 **ShoppingCart.jsx** 组件的代码。

▼ 点击这里查看 ShoppingCart.jsx 的代码

```
import React from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { removeItemFromCart, clearCart, increaseItemQuantity, decreaseItemQuantity } from './CartSlice'; // 假设您有增加和减少商品数量的
import './ShoppingCart.css'; // 导入组件特定样式的 CSS 文件
const ShoppingCart = () => {
  const dispatch = useDispatch();
  const cartItems = useSelector(state => state.cart.cartItems);
  const totalAmount = cartItems.reduce((total, item) => total + item.price * item.quantity, 0);
  const handleRemoveItem = itemId => {
    dispatch(removeItemFromCart(itemId));
  };
  const handleClearCart = () => {
    dispatch(clearCart());
  };
  const handleIncreaseQuantity = itemId => {
    dispatch(increaseItemQuantity(itemId));
  };
  const handleDecreaseQuantity = itemId => {
    dispatch(decreaseItemQuantity(itemId));
  };
  return (
    <>
      <div className="shopping-cart">
        <h2 className="shopping-cart-title">购物车</h2>
        <ul className="cart-items">
          {cartItems.map(item => (
            <li key={item.id} className="cart-item">
              <span>{item.name} - ${item.price}</span>
              <div className="quantity-controls">
                <button className="quantity-control-btn" onClick={() => handleDecreaseQuantity(item.id)}>-</button>
                <span> {item.quantity}</span>
                <button className="quantity-control-btn" onClick={() => handleIncreaseQuantity(item.id)}>+</button>
              </div>
              <button className="remove-item-btn" onClick={() => handleRemoveItem(item.id)}>Remove</button>
            </li>
          ))}
        </ul>
        <button className="clear-cart-btn" onClick={handleClearCart}>清空购物车</button>
      </div>
      <div>{totalAmount ? <div>'总金额为 {totalAmount}</div> : ''}</div>
    </>
  );
};
export default ShoppingCart;
```

第9步：检查输出

1. 在终端中通过按 `ctrl+c` 停止 React 应用程序的执行。
2. 然后，在终端中输入给定的命令并按 Enter。

   ```
   npm run preview
   ```

3. 要查看您的 React 应用程序，请刷新浏览器中已打开的 React 应用程序网页。如果没有打开，请点击左侧面板上的 Skills Network 图标。这将打开“SKILLS NETWORK TOOLBOX”。接下来，选择“启动应用程序”。在“应用程序端口”中输入端口号 **4173**，然后点击 。
4. 在将产品添加到购物车后，输出将如给定截图所示。



5. 再添加一个产品，您将看到总金额的变化。



请注意，添加到购物车 按钮只能使用一次来添加产品。之后，它将被禁用，如果您再次点击，它将不会添加相同的产品。

注意– 要查看最新更改，您需要在终端中再次执行 `npm run preview`。

恭喜！您已经创建了一个电子商务数据渲染的 React 应用程序！

步骤 10：实践任务

1. 现在在这个实践练习中，您将创建一个组件，在其中实现超级币的概念。
    - 超级币是一些电子商务平台或零售商作为其客户忠诚计划的一部分提供的一种忠诚或奖励积分。要查看用户根据总购物车金额赚取了多少，您需要创建此功能。
2. 为此，在选择 **Components** 文件夹后，右键单击创建一个名为 `SuperCoin.jsx` 的超级币组件。
3. 现在在组件的返回之前，使用 **useState** 钩子初始化 **superCoins** 变量及其对应的函数。

    提示：使用 **useState** 钩子将变量初始化为 `0`。

▼ 点击这里查看示例解决方案

```
const [superCoins, setSuperCoins] = useState(0);
```

4. 现在你需要使用 useSelector 钩子从 Redux 存储的状态中的购物车切片中检索 cartItems，以获取产品的总数量。

    提示：使用 **useSelector** 钩子来获取购物车商品的状态。

▼ 点击这里查看示例解决方案

```
const cartItems = useSelector(state => state.cart.cartItems);
```

5. 接下来，通过对 `cartItems` 数组中每个项目的价格和数量的乘积进行求和，使用 **reduce** 方法计算总金额。

    提示：使用 **useSelector** 钩子获取购物车项目的状态。

▼ 点击这里查看示例解决方案

```
const totalAmount = cartItems.reduce((total, item) => total + item.price * item.quantity, 0);
```

6. 现在您需要根据 totalAmount 更新 `superCoins` 状态：在不同的总金额范围内将其设置为 10、20 或 30 硬币，如果金额低于 100，则设置为 0。每当 totalAmount 变化时，此效果都会运行。

    提示：使用 **useEffect** 钩子来更新 superCoins 的状态。

▼ 点击这里查看示例解决方案

```
useEffect(() => {
  if (totalAmount >= 100 && totalAmount < 200) {
    setSuperCoins(10);
  } else if (totalAmount >= 200 && totalAmount < 300) {
    setSuperCoins(20);
  } else if (totalAmount >= 300) {
    setSuperCoins(30);
  } else {
    setSuperCoins(0);
  }
}, [totalAmount]);
```

7. 现在在函数组件的返回中使用jsx语法创建 ，并在 标签中集成superCoins变量状态。

   提示：使用{}来包含superCoins变量

▼ 点击这里查看示例解决方案

```
<div className="super-coins" style={{textAlign:'center'}}>
  <h2 className="super-coins-title">Super Coins</h2>
  <p className="super-coins-info">You will earn {superCoins} super coins with this purchase.</p>
</div>
```

8. 检查输出
   - 保存更改并重新运行应用程序。
   - 将产品添加到购物车中，当金额达到 **$100** 时，它将显示您已赚取 **10 超级币**。



9. 您可以增加金额，根据逻辑，超级币的价值也会增加。

结论

恭喜！您已经创建了一个电子商务 React 应用程序！

在这个实验中，您已经：

- 实现了 Redux Toolkit，以便在整个应用程序中进行统一的状态管理。
- 使用 React 和 Redux 开发了一个基本的电子商务平台。
- 为每个商品提供了一个“添加到购物车”按钮的商品列表。
- 使用户能够查看和管理添加到购物车的商品，包括删除商品。
- 利用 useDispatch 和 useSelector 钩子与 Redux 进行交互，提供全局数据访问。
- 确保了整个应用程序的无缝状态管理，提升了用户体验和可扩展性。

Author(s)

Richa Arora

© IBM Corporation 2023. 保留所有权利。