





14 Example: Shopping Site

In this example we'll build a simple shopping site that will demonstrate some of the things we've covered.

Here's what it will look like when we're done. It'll have a page full of items to buy:

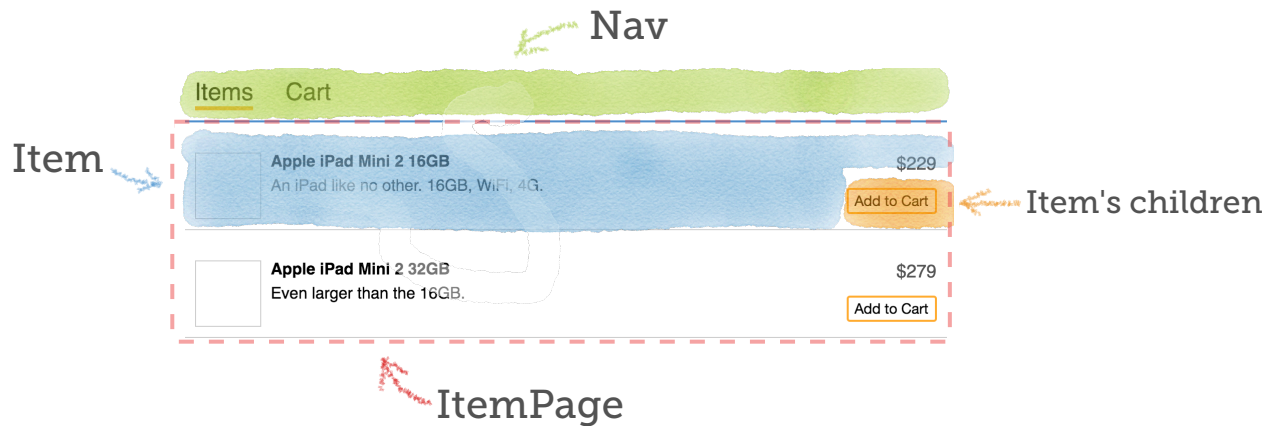
| Items | Cart |
|---|--|
|  | <div><div>Apple iPad Mini 2 16GB</div><div>An iPad like no other. 16GB, WiFi, 4G.</div><div>\$229</div><div>Add to Cart</div></div> |
|  | <div><div>Apple iPad Mini 2 32GB</div><div>Even larger than the 16GB.</div><div>\$279</div><div>Add to Cart</div></div> |

And it will have a shopping cart showing the selected items, with +/- buttons and a count for each one:

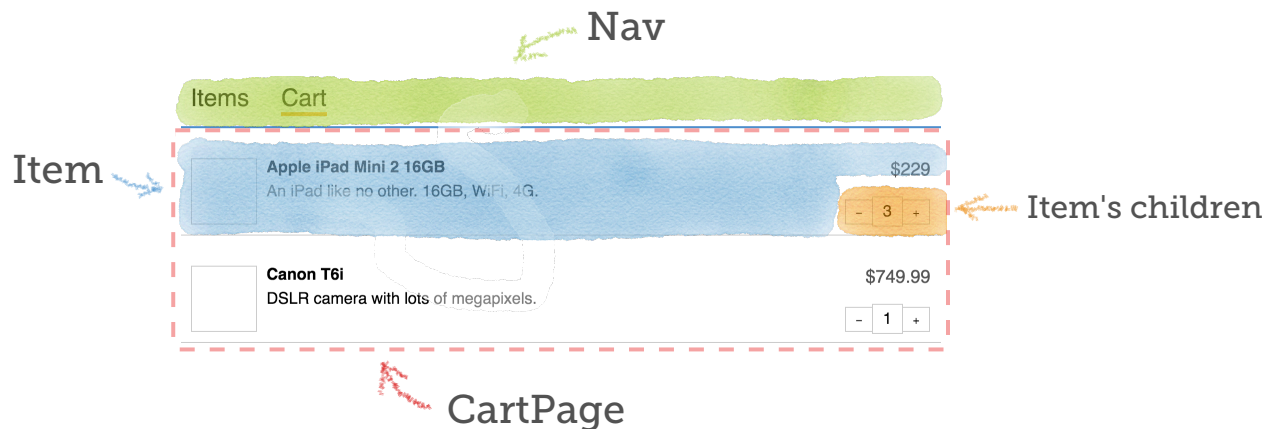
| Items | Cart |
|---|---|
|  | <div><div>Apple iPad Mini 2 16GB</div><div>An iPad like no other. 16GB, WiFi, 4G.</div><div>\$229</div><div><div>-</div><div>3</div><div>+</div></div></div> |
|  | <div><div>Canon T6i</div><div>DSLR camera with lots of megapixels.</div><div>\$749.99</div><div><div>-</div><div>1</div><div>+</div></div></div> |

Break Into Components

Following the same steps as before, let's start by breaking down the app into its components and giving them names. Here is the Items page:



And here is the Cart page:



There's also a top-level "App" component which contains everything else.

You'll notice that the two pages are very similar. Compared to previous examples, this one will have less-granular components. The components will have larger render functions with more "stuff" in them. You can always break things down later, and sometimes it's better to avoid abstracting into tiny pieces early on.

Here are the components we'll build:

- App
 - ItemPage
 - CartPage
 - Item
 - Nav

Start Building

We'll start by generating a new project:

```
$ create-react-app shopper && cd shopper
```

Open up `src/App.js` and replace its contents with this to get started:

```
import React from 'react';
import Nav from './Nav';
import './App.css';

class App extends React.Component {
  renderContent() {
    return (
      <span>Empty</span>
    );
  }

  render() {
    return (
      <div className="App">
        <Nav/>
        <main className="App-content">
          {this.renderContent()}
        </main>
      </div>
    );
  }
}
```

```
}  
  
export default App;
```

Then create `src/Nav.js` and start it off with this:

```
import React from 'react';  
  
const Nav = () => (  
  <nav className="App-nav">  
    <ul>  
      <li className="App-nav-item"><a>Items</a></li>  
      <li className="App-nav-item"><a>Cart</a></li>  
    </ul>  
  </nav>  
>);  
  
export default Nav;
```

Start up the app and see what we have:

```
$ yarn start
```

- Items
- Cart

Empty

It's working, but it needs styling. Open up `src/App.css`, and replace the contents with this:

```
.App {
  max-width: 800px;
  margin: 0 auto;
}
.App-nav {
  margin: 20px;
  padding: 10px;
  border-bottom: 2px solid #508FCA;
}
.App-nav ul {
  padding: 0;
  margin: 0;
}
.App-nav-item {
  list-style: none;
  display: inline-block;
  margin-right: 32px;
  font-size: 24px;
  border-bottom: 4px solid transparent;
}
.App-content {
  margin: 0 20px;
}
```

That's a little better:

Items Cart

Empty

Next, we need to keep track of the active tab, and choose whether to render `ItemPage` or `CartPage`. To do this, we'll store the active tab in state, and make the `Nav` trigger a state change. The `App` component is the best home for this state, since `App` will need it to know which page to render.

Initialize the state at the top, and add a function to select a tab:

```
class App extends Component {
  state = {
    activeTab: 0
  };

  handleTabChange = (index) => {
    this.setState({
      activeTab: index
    });
  }

  // ...
}
```

Now, we need to pass the `handleTabChange` function down into `Nav`, along with the `activeTab` so it can render correctly. We'll also update `renderContent` to display the active page.

```
class App extends Component {
  // ...

  renderContent() {
    switch(this.state.activeTab) {
      default:
      case 0: return <span>Items</span>;
      case 1: return <span>Cart</span>;
    }
  }

  render() {
    let {activeTab} = this.state;
    return (
      <div className="App">
        <Nav activeTab={activeTab} onTabChange={this.handleTabChange} />
        <main className="App-content">
          {this.renderContent()}
        </main>
      </div>
    );
  }
}
```

```
    </div>
  );
}
}
```

Update the Nav component in `src/Nav.js` to use the new props:

```
const Nav = ({ activeTab, onTabChange }) => (
  <nav className="App-nav">
    <ul>
      <li className={`App-nav-item ${activeTab === 0 && 'selected'} `}>
        <a onClick={() => onTabChange(0)}>Items</a>
      </li>
      <li className={`App-nav-item ${activeTab === 1 && 'selected'} `}>
        <a onClick={() => onTabChange(1)}>Cart</a>
      </li>
    </ul>
  </nav>
);
```

Then add some styles to the bottom of `src/App.css`:

```
.App-nav-item a {
  cursor: pointer;
}
.App-nav-item a:hover {
  color: #999;
}
.App-nav-item.selected {
  border-bottom-color: #FFAA3F;
}
```

Here's what it should look like now:

Items Cart

Cart

Arrow Function onClick

You'll notice that we're passing an arrow function to the `onClick` handlers here. At first glance this may look strange. You might think `onClick={onTabChange(0)}` looks better.

But remember, props are evaluated *before* they're passed down. When written without the arrow function, `onTabChange` would be called *when Nav renders*, instead of when the link is clicked. That's not what we want. Wrapping it in an arrow function delays execution until the user clicks the link.

You're bound to forget to wrap a click handler in an arrow function in one of your own projects some day. I've done it many times. If you ever find yourself wondering, "Why is this handler running early?!" – check how it's being passed.

Creating Functions in Render: Bad?

Another note about this: it's generally a bad idea to create new functions to pass into props like the code above, but it's worth understanding why, so you can apply the rule when it makes sense.

The reason it's bad to create new functions is for performance: not because creating a function is expensive, but because passing a new function down to a *pure* component every time it renders means that it will always see "new" props, and therefore always re-render (needlessly).

Avoid creating a new function in render when (a) the child component receiving the function prop is *pure* and (b) you expect the parent component to re-render often.

Function components and classes that extend `Component` are not pure. They will *always* re-render, even if their props haven't changed. If a class extends `PureComponent` instead, though, it *is* pure and it will skip re-rendering when its props are unchanged. Avoiding needless re-renders is the easiest way to improve performance in a React app.

That said, it's important to consider how often this component will re-render, and also how painful it will be to avoid creating that function every time. Sometimes it's just not worth the effort. You can always profile the app and improve performance later; sometimes it makes the most sense to do the quick & easy thing now, get it working, and come back later if it proves to be slow.

The Nav component above will probably not re-render very often, but for the sake of example, how would you get rid of the arrow functions? Here's one way to do it:

```
const Nav = ({ activeTab, onTabChange }) => (
  <nav className="App-nav">
    <ul>
      <li className={`App-nav-item ${activeTab === 0 && 'selected'}`}>
        <NavLink index={0} onClick={onTabChange}>Items</NavLink>
      </li>
      <li className={`App-nav-item ${activeTab === 1 && 'selected'}`}>
        <NavLink index={1} onClick={onTabChange}>Cart</NavLink>
      </li>
    </ul>
  </nav>
);

class NavLink extends React.Component {
  handleClick = () => {
    this.props.onClick(this.props.index);
  }

  render() {
    return (
      <a onClick={this.handleClick}>
        {this.props.children}
      </a>
    );
  }
}
```

Here, the handleClick function will only be created *once*, when the component is rendered the first time. This fixes the problem (render no longer creates a function), but at the expense of more complex code (a whole new component!). It's a tradeoff, and one worth considering when you run into this with your own projects.

Create ItemPage

Let's turn our attention to the list of items for sale. We'll create a new component to display this data. Create a file called `src/ItemPage.js` and type this in:

```
import React from 'react';
import PropTypes from 'prop-types';
import './ItemPage.css';

function ItemPage({ items }) {
  return (
    <ul className="ItemPage-items">
      {items.map(item =>
        <li key={item.id} className="ItemPage-item">
          {item.name}
        </li>
      )}
    </ul>
  );
}
ItemPage.propTypes = {
  items: PropTypes.array.isRequired
};

export default ItemPage;
```

Nothing new here: we're just iterating over a list of items and rendering them out. Since the individual items are going to be more involved than just a "name", we'll extract that out into its own component shortly.

This component relies on `ItemPage.css`, which doesn't exist yet. Create that file, and add this CSS to style the list:

```
.ItemPage-items {
  margin: 0;
  padding: 0;
}
.ItemPage-items li {
```

```
list-style: none;
margin-bottom: 20px;
}
```

We're going to need some items to sell. We'll use static data here, same as before.

I like to start with static data even when I have a server that can return real data. You can go from *nothing* to *something* pretty quickly when there are no moving parts. And honestly, it's just more *fun* when there's something on the screen that I can tweak.

Create a file called `static-data.js` (under `src`) and paste in this code.

```
let items = [
  {
    id: 0,
    name: "Apple iPad Mini 2 16GB",
    description: "An iPad like no other. 16GB, WiFi, 4G.",
    price: 229.00
  },
  {
    id: 1,
    name: "Apple iPad Mini 2 32GB",
    description: "Even larger than the 16GB.",
    price: 279.00
  },
  {
    id: 2,
    name: "Canon T7i",
    description: "DSLR camera with lots of megapixels.",
    price: 749.99
  },
  {
    id: 3,
    name: "Apple Watch Sport",
    description: "A watch",
    price: 249.99
  },
  {
```

```
    id: 4,  
    name: "Apple Watch Silver",  
    description: "A more expensive watch",  
    price: 599.99  
  }  
];  
  
export {items};
```

Then we will pass those items into the new ItemPage component.

Back in App.js, import the ItemPage and the static items data.

```
import ItemPage from './ItemPage';  
import {items} from './static-data';
```

And then update the renderContent function to use the new ItemPage:

```
class App extends Component {  
  // ...  
  renderContent() {  
    switch(this.state.activeTab) {  
      default:  
        case 0: return <ItemPage items={items}/>  
        case 1: return <span>Cart</span>;  
    }  
  }  
  // ...  
}
```

Now you should see some items rendering!

Items Cart

Apple iPad Mini 2 16GB

Apple iPad Mini 2 32GB

Canon T7i

Apple Watch Sport

Apple Watch Silver

Who Owns the Data?

In your own apps, you'll get to decide which components "own" which pieces of data. Here, we're saying that App owns the list of items and the state of the cart, and it passes them down to the relatively dumb `ItemPage` (and soon, `CartPage`). When it came time to use *real* data from a server, we'd fetch the data from within App.

In this case, because the items are needed by both pages, we pulled the data up to App. But sometimes it'll make more sense for the page-level components to own the data. For instance, if you had an `ItemDetail` page that wanted to display a list of reviews for an item, it would probably make the most sense for `ItemDetail` to "own" that review data.

Create Item

Let's now create the real `Item` component to use in `ItemList`. It'll take an `item` prop, and we'll also want to be able to add it to the cart, so it should take an `onAddToCart` prop too.

This component will be stateless. Its responsibilities are to display an item, and to let its parent know when the "Add to Cart" button is clicked.

Learning from the GitHub example, we won't return an `` from this component. We don't want to force users of `Item` to put it inside a ``. By returning a plain `<div>`, the parent can put this item into an `` (or not) if it chooses to. This keeps the code more flexible.

With that in mind, create `src/Item.js` and `src/Item.css`. In `Item.js`, write out the component:

```
import React from 'react';
import PropTypes from 'prop-types';
import './Item.css';

const Item = ({ item, onAddToCart }) => (
  <div className="Item">
    <div className="Item-left">
      <div className="Item-image" />
      <div className="Item-title">
        {item.name}
      </div>
      <div className="Item-description">
        {item.description}
      </div>
    </div>
    <div className="Item-right">
      <div className="Item-price">
        ${item.price}
      </div>
      <button
        className="Item-addToCart"
        onClick={onAddToCart}
      >
        Add to Cart
      </button>
    </div>
  </div>
);
Item.propTypes = {
  item: PropTypes.object.isRequired,
  onAddToCart: PropTypes.func.isRequired
};

export default Item;
```

Now let's wire it in to the `ItemPage` component. In `ItemPage.js`, update the code to look like this:

```
import React from 'react';
import PropTypes from 'prop-types';
import Item from './Item';
import './ItemPage.css';

function ItemPage({ items, onAddToCart }) {
  return (
    <ul className="ItemPage-items">
      {items.map(item =>
        <li key={item.id} className="ItemPage-item">
          <Item
            item={item}
            onAddToCart={() => onAddToCart(item)} />
        </li>
      )}
    </ul>
  );
}

ItemPage.propTypes = {
  items: PropTypes.array.isRequired,
  onAddToCart: PropTypes.func.isRequired
};

export default ItemPage;
```

We're importing `Item`, and then inside the `` we're now rendering the `Item` component instead of just the item's name. Also new is the `onAddToCart` prop and its associated `propTypes`.

At this point, the code won't work, and there will be a warning about the missing `onAddToCart` prop.

Now that `ItemPage` requires an `onAddToCart` prop, we need to pass that from inside the `App` component. While we're at it, we may as well actually add the items to a cart! We can store that in state, and we will add a new `cart` array to hold the items.

Let's walk through it. First, in `App.js`, add a `cart` property to the initial state, and initialize it to an empty array:

```
class App extends Component {
  state = {
    activeTab: 0,
    cart: []
  }

  // ...
}
```

Then update `renderContent` to pass a function as the `onAddToCart` prop:

```
class App extends Component {
  // ...

  renderContent() {
    switch(this.state.activeTab) {
      default:
      case 0:
        return (
          <ItemPage
            items={items}
            onAddToCart={this.handleAddToCart} />
        );
      case 1:
        return <span>Cart</span>;
    }
  }

  // ...
}
```


The function `this.handleAddToCart` doesn't exist yet, so create that next. It will accept an item, and add the item to the cart:

```
class App extends Component {  
  // ...  
  
  handleAddToCart = (item) => {  
    this.setState({  
      cart: [...this.state.cart, item.id]  
    });  
  }  
  
  // ...  
}
```

What this is doing is setting the cart state to a copy of the current cart, plus one new item.

There's some new ES6 syntax here: `...this.state.cart` is the *spread* operator, and it expands the given array into its individual items. Here's an example of the spread operator:

```
// With arrays:  
var a = [1, 2, 3];  
var b = [a, 4];    // => [[1, 2, 3], 4]  
var c = [...a, 4]; // => [1, 2, 3, 4]  
  
// With objects (technically not ES6)  
var o1 = {a: 1, b: 2};  
var o2 = {...o1, c: 3}; // => {a: 1, b: 2, c: 3}
```

Note that the spread operator for arrays is officially part of ES6, but the one for objects is not. It will likely be officially added to JS in the future. For now, object spread is supported by Babel, which Create React App is using under the hood to turn our code into browser-compatible ES5.

Updating State Immutably

You might be wondering... why not just modify state directly and then call `setState`, like this?

```
this.state.cart.push(item.id);  
this.setState({cart: this.state.cart});
```

This is a bad idea for two reasons (even though it would work in this example).

You should *never* modify (“mutate”) state or its child properties directly. Don’t do `this.state.something = x`, and don’t do `this.state = x`. React relies on you to call `this.setState` when you want to make a change, so that it will know something changed and trigger a re-render. If you circumvent `setState` the UI will get out of sync with the data.

Mutating state directly can lead to odd bugs, and components that are hard to optimize.

Here’s an example. Recall that one of the common ways to make a React component more performant is to make it “pure,” which causes it to only re-render when its props change (instead of every time its parent re-renders). This can be done automatically by extending `React.PureComponent` instead of `React.Component`, or manually by implementing the `shouldComponentUpdate` lifecycle method to compare `nextProps` with current props. If the props look the same, it skips the render, and saves some time.

Here is a simple component that renders a list of items. Notice that it extends `React.PureComponent`:

(If you have the example code, this code is under the “examples/impure-state” folder.)

```
class ItemList extends React.PureComponent {  
  static propTypes = {  
    items: PropTypes.array.isRequired  
  }  
  
  render() {  
    return (  
      <ul>  
        {this.props.items.map(item =>  
          <li key={item.id}>{item.value}</li>  
        )}  
      )  
    )  
  }  
}
```

```
    </ul>
  );
}
}
```

Now, here is a tiny app that renders the `ItemList` and allows you to mutably or immutably add items to the list:

```
class App extends Component {
  // Initialize items to an empty array
  state = {
    items: []
  }

  // Initialize a counter that will increment
  // for each item ID
  nextItemId = 0;

  makeItem() {
    // Create a new ID and use
    // a random number as the value
    return {
      id: this.nextItemId++,
      value: Math.random()
    };
  }

  // The Right Way:
  // copy the existing items and add a new one
  addItemImmutably = () => {
    this.setState({
      items: [
        ...this.state.items,
        this.makeItem()
      ]
    });
  }
}
```

```
// The Wrong Way:
// mutate items and set it back
addItemMutably = () => {
  this.state.items.push(this.makeItem());
  this.setState({ items: this.state.items });
}

render() {
  return (
    <div>
      <button onClick={this.addItemImmutably}>
        Add item immutably (good)
      </button>
      <button onClick={this.addItemMutably}>
        Add item mutably (bad)
      </button>
      <ItemList items={this.state.items}/>
    </div>
  );
}
```

Try it out. Click the immutable Add button a few times and notice how the list updates as expected. Then click the mutable Add button and notice how the new items don't appear, even though state is being changed.

This happens because `ItemList` is *pure*, and because pushing a new item on the `this.state.items` array does not replace the underlying array. `ItemList` will notice that its props haven't changed and it will not re-render.

Finally, click the immutable Add button again, and watch how the `ItemList` re-renders with all the missing (mutably-added) items.

That is a long way of saying that you shouldn't mutate state, even if you immediately call `setState`. Optimized components (ones that implement `shouldComponentUpdate`) might not re-render if you do.

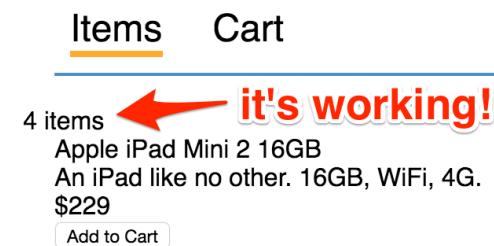
Instead, you should create *new* objects and arrays when you call `setState`, which is what we did above with the spread operator.

Back to the Code

Let's try it now: clicking "Add to Cart" should update the cart. But we have no way to verify the cart is filling up! Just for debugging, add this bit of code inside `App.js`'s render method:

```
<div>
  {this.state.cart.length} items
</div>
```

Now try it again. The item count should increase each time you click "Add to Cart." Once you verify that it's working, you can remove that debugging code.



Let's add some CSS to make it look a little better. In `Item.css` (which you created earlier, but is currently empty), add this CSS:

```
.Item {
  display: flex;
  justify-content: space-between;
  border-bottom: 1px solid #ccc;
  padding: 10px;
}
.Item-image {
  width: 64px;
  height: 64px;
  border: 1px solid #ccc;
  margin-right: 10px;
  float: left;
}
.Item-title {
  font-weight: bold;
```

```
    margin-bottom: 0.4em;
}
.Item-price {
    text-align: right;
    font-size: 18px;
    color: #555;
}
.Item-addToCart {
    margin-bottom: 5px;
    font-size: 14px;
    border: 2px solid #FFAA3F;
    background-color: #fff;
    border-radius: 3px;
    cursor: pointer;
}
.Item-addToCart:hover {
    background-color: #FFDDB2;
}
.Item-addToCart:active {
    background-color: #ED8400;
    color: #fff;
    outline: none;
}
.Item-addToCart:focus {
    outline: none;
}
.Item-left {
    flex: 1;
}
.Item-right {
    display: flex;
    flex-direction: column;
    justify-content: space-between;
}
```

That's a bit better:

Items Cart



Apple iPad Mini 2 16GB

An iPad like no other. 16GB, WiFi, 4G.

\$229

Add to Cart



Apple iPad Mini 2 32GB

Even larger than the 16GB.

\$279

Add to Cart

Create CartPage

Now that we can add items to the cart, we'll build the component that renders the cart itself.

Create `src/CartPage.css` and `src/CartPage.js`, and open up `CartPage.js`. Initially, the code will be very similar to the code from `ItemPage`.

```
import React from 'react';
import PropTypes from 'prop-types';
import Item from './Item';
import './CartPage.css';

function CartPage({ items }) {
  return (
    <ul className="CartPage-items">
      {items.map(item =>
        <li key={item.id} className="CartPage-item">
          <Item item={item} />
        </li>
      )}
    </ul>
  );
}

CartPage.propTypes = {
  items: PropTypes.array.isRequired
}
```

```
};  
  
export default CartPage;
```

CartPage accepts a list of items and renders the items as an unordered list. You'll notice that we aren't passing an `onAddToCart` handler to `Item`, which will cause a prop warning. We'll fix that shortly.

Pass Items to CartPage

The cart needs items to display. The trouble is, our cart state just contains item indices, not actual items. Additionally, there can be duplicate indices when an item is added multiple times. We'll need to massage the cart state into an array of items suitable to pass into `CartPage`.

In `App.js`, import `CartPage` at the top:

```
import CartPage from './CartPage';
```

Then add a `renderCart` function, and update `renderContent` to call it.

```
class App extends Component {  
  // ...  
  
  renderContent() {  
    switch(this.state.activeTab) {  
      default:  
      case 0:  
        return (  
          <ItemPage  
            items={items}  
            onAddToCart={this.handleAddToCart} />  
        );  
      case 1:  
        return this.renderCart();  
    }  
  }  
}
```



```
renderCart() {  
  // Count how many of each item is in the cart  
  let itemCounts = this.state.cart.reduce((itemCounts, itemId) => {  
    itemCounts[itemId] = itemCounts[itemId] || 0;  
    itemCounts[itemId]++;  
    return itemCounts;  
  }, {});  
  
  // Create an array of items  
  let cartItems = Object.keys(itemCounts).map(itemId => {  
    // Find the item by its id  
    var item = items.find(item =>  
      item.id === parseInt(itemId, 10)  
    );  
  
    // Create a new "item" and add the 'count' property  
    return {  
      ...item,  
      count: itemCounts[itemId]  
    }  
  });  
  
  return (  
    <CartPage items={cartItems} />  
  );  
}  
  
// ...  
}
```

There are a couple things in that code you may not have seen before.

Array.prototype.reduce

In the code above, we're using Array's built-in reduce function to build up an object where the keys are itemIds and the values are the number of each item in the cart.

`reduce` is a handy general-purpose function that takes an array and boils it down to a single result. In other words it “reduces” the array to a single value. In the example above, that’s an object. In the one below, it’s a number. It can be whatever you want.

`reduce` takes a function as an argument, and it calls your provided function once for each element of the array, similar to Array’s `map`. Unlike `map`, though, your function gets called with 2 arguments: the previous iteration’s result, and the current array element. And the value you return matters: it becomes the new “result” and is fed into the next function call.

The first time it is called, since the “previous iteration” hasn’t happened yet, it needs an initial value which we supplied as the last argument above (the empty object `{}`). The initial value is optional though. If you don’t pass one, `reduce` will use the array’s first element as the initial value.

Here’s a smaller, simpler example of `reduce` in action, with a step-by-step list of what happens:

```
let a = [1, 2, 3, 4];
let total = a.reduce((sum, value) => {
  return sum + value;
}, 0);

// The arrow function is called 4 times:
//   (0, 1) => returns 1
//   (1, 2) => returns 3
//   (3, 3) => returns 6
//   (6, 4) => returns 10
// Then reduce returns the last return value (10)
```

The `reduce` function is a functional-style shorthand for code like this:

```
const a = [1, 2, 3, 4];
let total = 0;
for(let i = 0; i < a.length; i++) {
  total += a[i];
}
```

Any time you want to iterate over the values of an array to create an aggregate result, consider using `reduce`.

Object.keys



`Object.keys` returns an array of the keys in an object. For instance `Object.keys({a: 1, b: 2})` would return `['a', 'b']`. Objects don't have a built-in iterator `forEach` function like arrays do, so `Object.keys` makes it easier to iterate over the keys of an object.

Modifying the Cart

At this point, the “Cart” tab should be working. Click “Add to Cart” a few items, then click over to the “Cart” tab, and the items should be there.

However, the “Add to Cart” buttons are out of place on the Cart page, and we're still getting propTypes warnings about `onAddToCart`. We'll fix both of those next.

Items Cart

| | | |
|---|---|--------------------------------------|
|  | Apple iPad Mini 2 16GB An iPad like no other. 16GB, WiFi, 4G. | \$229 Add to Cart |
|  | Apple iPad Mini 2 32GB Even larger than the 16GB. | \$279 Add to Cart |

Even though the cart is working, it's missing important features. There is no way to add or remove items, no total price, and we don't even know how many of each item is in the cart.

Remember how we reused the `Item` component inside `CartPage`? It would be nice if we could render a customized `Item` that had Add and Remove buttons in place of the “Add to Cart” button.

There are a few ways to go about this.

One idea: we could make a copy of the `Item` component, rename it `CartItem`, and customize it as necessary. This is quick and easy, but duplicates code.

Another idea: we could make `Item` take a prop that tells it to be a “regular item” or a “cart item.” Using that prop, `Item` would decide whether to render an “Add to Cart” button or “Add/Remove” buttons. This would work, but it’s messy – `Item` would have multiple responsibilities. If it became necessary to reuse `Item` in a third situation, we might end up adding even more conditional logic, making `Item` harder to understand and maintain.

A third idea is to pass children to `Item` and let it decide where to put them. This is the idea we’ll implement. It makes `Item` fairly reusable. If you want an “Add to Cart” button, pass that in. If you want Add/Remove buttons instead, pass those in. Let’s see how this works.

Refactoring Item

The new `Item` and its `propTypes` looks like this:

```
const Item = ({ item, children }) => (  
  <div className="Item">  
    <div className="Item-left">  
      <div className="Item-image" />  
      <div className="Item-title">{item.name}</div>  
      <div className="Item-description">{item.description}</div>  
    </div>  
    <div className="Item-right">  
      <div className="Item-price">${item.price}</div>  
      {children}  
    </div>  
  </div>  
</div>  
);  
Item.propTypes = {  
  item: PropTypes.object.isRequired,  
  children: PropTypes.node  
};
```

Where once there was an “Add to Cart” `<button>`, we now have `{children}` instead. Also, `Item` no longer needs to know about the `onAddToCart` function, so we’ve removed its `propTypes`.

If you refresh at this point, you’ll see that the “Add to Cart” buttons are gone from the Cart page *and* the Items page. To fix that, open up `ItemPage.js` and pass a `<button>` as a child of the `Item`:

```
function ItemPage({ items, onAddToCart }) {
  return (
    <ul className="ItemPage-items">
      {items.map(item =>
        <li key={item.id} className="ItemPage-item">
          <Item item={item}>
            <button
              className="Item-addToCart"
              onClick={() => onAddToCart(item)}>
              Add to Cart
            </button>
          </Item>
        </li>
      )}
    </ul>
  );
}
```

There we go, back to normal. Now for the new part: open `CartPage.js`, and inside `Item`, pass in the plus/minus buttons and the item count. We're also going to need handler functions for when an item is added or removed.

```
function CartPage({ items, onAddOne, onRemoveOne }) {
  return (
    <ul className="CartPage-items">
      {items.map(item =>
        <li key={item.id} className="CartPage-item">
          <Item item={item}>
            <div className="CartItem-controls">
              <button
                className="CartItem-removeOne"
                onClick={() => onRemoveOne(item)}>&ndash;</button>
              <span className="CartItem-count">{item.count}</span>
              <button
                className="CartItem-addOne"
                onClick={() => onAddOne(item)}>+</button>
            </div>
          </Item>
        </li>
      )}
    </ul>
  );
}
```

```

        </Item>
      </li>
    )}
  </ul>
);
}
CartPage.propTypes = {
  items: PropTypes.array.isRequired,
  onAddOne: PropTypes.func.isRequired,
  onRemoveOne: PropTypes.func.isRequired
};

```

This is all stuff you've seen before.

The code won't work until we pass the `onAddOne` and `onRemoveOne` functions, so head let's back to `App.js` and create those. At the bottom of `renderCart`, we need to pass the extra props to `CartPage`:

```

class App extends Component {
  // ...
  renderCart() {
    // ...
    return (
      <CartPage
        items={cartItems}
        onAddOne={this.handleAddToCart}
        onRemoveOne={this.handleRemoveOne} />
    );
  }
  // ...
}

```

We already have the `handleAddToCart` function that takes an item and adds its id to the cart, so we can just reuse that here.

We need to create the `handleRemoveOne` function. It will take an item, find an occurrence of that item in the cart, and then update the cart state to remove that occurrence.

```
class App extends Component {
  // ...
  handleRemoveOne = (item) => {
    let index = this.state.cart.indexOf(item.id);
    this.setState({
      cart: [
        ...this.state.cart.slice(0, index),
        ...this.state.cart.slice(index + 1)
      ]
    });
  }
  // ...
}
```

We're using the array *spread* operator twice here, in order to avoid mutating the state. We're taking the left half of the array (up to *but not including* the item we want to remove) and concatenating it with the right half of the array (everything after the item being removed).

This is a little more work than if we were to allow mutations but it is a good habit to get into. It'll be especially important if you start using Redux in the future. Redux relies heavily on immutability.

To wrap up the add/remove buttons, let's give them a bit of style. Add this CSS to `CartPage.css`:

```
.CartPage-items {
  margin: 0;
  padding: 0;
}
.CartPage-items li {
  list-style: none;
}
.CartItem-count {
  padding: 5px 10px;
  border: 1px solid #ccc;
}
.CartItem-addOne,
.CartItem-removeOne {
  padding: 5px 10px;
```

```
border: 1px solid #ccc;
background: #fff;
}
.CartItem-addOne {
border-left: none;
}
.CartItem-removeOne {
border-right: none;
}
```

Here it is in all its glory:

Items Cart



Apple iPad Mini 2 16GB

An iPad like no other. 16GB, WiFi, 4G.

\$229

- 2 +



Canon T6i

DSLR camera with lots of megapixels.

\$749.99

- 2 +

Exercises

1. It would be nice if the shopping cart told us how many thousands of dollars we were about to spend on Apple products, rather than letting us guess. Add a “Total” to the bottom of the Cart page similar to this:

| | | |
|---|---|---------------------------|
|  | Apple iPad Mini 2 32GB Even larger than the 16GB. | \$279 <div>- 4 +</div> |
|---|---|---------------------------|

Total: \$2032

2. The Cart page is completely blank when the cart is empty. Modify it to display “Your cart is empty” when there are no items in the cart, something like this:

Items Cart


Your cart is empty.

Why not add some expensive products to it?

3. Display a summary of the shopping cart in the top-right of the nav bar, as shown below. Include the total number of items in the cart and the total price. Make sure to account for the fact that each item in the cart array could have a count greater than 1. As a bonus, make the cart summary clickable, and switch to the Cart page on click. Add a shopping cart icon too if you like (this one is from Font Awesome).

Items Cart

 2 items (\$458)

| | | |
|---|---|---------------------------|
|  | Apple iPad Mini 2 16GB An iPad like no other. 16GB, WiFi, 4G. | \$229 <div>- 2 +</div> |
|---|---|---------------------------|