- ⏱ 2 hours

Explore React Hooks

Another developer started a product list page and has requested your help to complete it.

Here are the incomplete features.

- Click button below each item to see its details in the side panel
- Show highlight on item itself (so user can see the current selection)
- Allow user to open/close the side panel
- Automatically open the side panel if it's closed when an item is clicked
- Automatically clear the selection when the side panel is closed

Go ahead and clone the starter from the `Download` link at the bottom of this page. All the changes you need to make will be in *src/components/ProductView/ProductView.js*.

Prepare

Before starting to change code on a new-to-you project, it is helpful to become familiar with what's already been done. In this case, the original developer used create-react-app as the starting point, then added a few components to display a product list. They determined that the product data will come into the view through a prop named `products`. (It does not matter where that data comes from as that will be managed outside the component.) Additionally, they finished the layouts in the UI with a box, image, and button for each product as well as a panel on the right side.
When you run the application (`npm install` followed by `npm start`), you'll see these UI elements display, but clicking does not do anything. That's where you come in!
When you look at the code in `ProductView`, you'll find `console.log` statements in the `onClick` event handlers. That's a clue some work is needed there. Additionally, you'll find a comment that starts with "TODO". While that might not be the only work required, these will be elements you need to modify.

An astute developer may also notice there are some props on other components that aren't in use yet. The extra effort to find them while you're coming up to speed may save you some time later since you may be able to leverage them to accomplish your tasks.

Next, you need to pick somewhere to start. The panel toggle seems good since it has a `TODO` comment.

Phase 1: Toggle side panel

To replace a constant with a state variable, you'll want to employ the `useState` hook. Perhaps, something like this

```
const [sideOpen, setSideOpen] = useState(true);
```

Remember to add its import at the top. It will come from the `react` library.
The application should still run after this change, although VS Code and/or the JavaScript console will probably warn you that `setSideOpen` is defined but not used.
Go ahead and use it to replace the `console.log` associated with the toggle UI on the side panel. Remember

- State variables, like constants, are accessible within the JSX.
- Likewise, the setting function for any state variable can be called from JSX.
- Setting a boolean value that's `true` to `false` (or `false` to `true`) is the purpose of the NOT operator (`!`).

All this means you could choose to toggle the value with a function call like `setSideOpen(!sideOpen)`.
After replacing the `console.log` you will likely have code that looks something like this:

```
<div className="product-side-panel-toggle"
     onClick={() => setSideOpen(!sideOpen)}>
```

If all the changes are correct, you will be able to click the ">" tab on the side panel to close it. Study the code and see if you can find where it switches to "<". It is common practice in **React** to use a state variable in multiple UI updates.

Phase 2: Track selected item

Now, you will add a second state variable on your own. This one will track the `selectedProduct`.
Hint: The previous developer left a `console.log` in the handler for `onClick` of each `ProductListItem` to show you the data that is needs to be stored in state for the selected item.

In order for the info to show in the side panel, the new state variable will need to be passed through a prop to the ProductDetails component. Go ahead and add that as well.

Hint: The `visible` prop is already receiving the value of the `showOpen` state variable. If you look at the ProductDetails component, you will find the other prop that the previous developer already coded.

When all is connected correctly, you can run the application, click each button below the images, and see associated details in the right side panel.

What would help the user experience is to also highlight the box in the list for the item whose details are showing. Again, the previous developer thought of this and created a prop to change the style, `isSelected`.

Think about the condition for showing the selection and put that into the JSX.

HINT: Both `item` and `selectedProduct` are objects that have "id" attributes.

Finally, you can run the application again to see both the highlight on the list and the side panel both update.

Phase 3: Enhance the user experience

Congratulations getting two state variables working! You'll notice as you click around the application, these variables are completely independent. This can leave the user with a disconnect where they see an item highlighted in the list without also seeing the details.

For example, if you close the side panel then click the buttons, you will not be able to see the details immediately. You have to take an extra step to open the side panel. A nicer user experience would be to open the side panel automatically when the details show.

Likewise, if you close the side panel while looking at an item's details, the highlight still shows in the list. Users may appreciate it more if the selection automatically clears when the side panel is closed.

Fortunately, **React** hooks can help in both situations. Specifically, the `useEffect` hook can help. This hook works by calling a function when one or more variables change. These variables can be either passed in through props or managed with state. In this component, you have two state variables to work with.

Auto-open side panel

First, consider which variable change will trigger the auto-open effect (`selectedProduct`) and which setter function will need to be called (`setSideOpen(true)`).
Second, code the framework of the `useEffect` hook. Each time you utilize `useEffect`, it can start this same way.

```
useEffect(() => {
}, []);
```

Finally, put the trigger into the array and the call to the setter in the function body. Consider whether or not you need any conditionals (you might not).

You probably ended up with something like this

```
useEffect(() => {
    setSideOpen(true);
}, [selectedProduct]);
```

Try it out in the browser by refreshing, closing the side panel, and clicking one of the buttons in the list. If all is well, the side panel will open while showing that item's details.

Clear the selection when the panel closes

Again, consider which variable change will trigger the effect (`sideOpen`), and which setter function will need to be called (`setSelection`).
Second, put in the framework for `useEffect`. (You can find it above, if you'd like to look at it again.)

Finally, put the trigger into the array and the call to the setter in the function body. Again, consider what conditionals are needed, if any.

You probably wrote something like this.

```
useEffect(() => {
    setSelectedProduct();
}, [sideOpen]);
```

Refresh your browser and try to close the side panel. One of three things will happen:

1. If all worked as expected, congratulations! You are ready to continue with the next phase.
2. If the highlight in the list disappears, and the side panel resets to "Our Products..." while remaining open, that's okay. Continue to the next phase to do some debugging, and you can solve this bug in the process.
3. If the list did not clear or other errors occurred, please check your code. You can take a peek at the next phase to see if debugging state change will solve your issues. Good developers know when to reach out for help, if necessary, before continuing.

Phase 4: Debug re-rendering

The first step is understanding when the values of state variables have changed. This means combining `console.log` with `useEffect`.
The next step is to look at the components themselves, the one with the `useState` hooks as well as its children.

Debugging state change

Begin by adding `console.log`s to both `useEffect` functions so you can see when they are being triggered. Many developers like to start with the logging as the first statement, i.e., before any conditionals they might have.
For example, if you had the bug with side panel resetting but not closing, you should now have `useEffect` hooks that look something like this:

```
    // Open side panel
    useEffect(() => {
        console.log(`selectedProduct CHANGED TO`,
selectedProduct);
        setSideOpen(true);
    }, [selectedProduct]);

    // Deselect product
    useEffect(() => {
        console.log(`sideOpen CHANGED TO`, sideOpen);
        setSelectedProduct();
    }, [sideOpen]);
```

When you watch the console in the browser, you'll see output like this:

```
// After refresh
selectedProduct CHANGED TO undefined
sideOpen CHANGED TO true

// Select first item (abbreviated output)
selectedProduct CHANGED TO {id: 1, code: "abc", name: "Bistro Table
...

// Close side panel
sideOpen CHANGED TO false
ProductView.js:20 selectedProduct CHANGED TO undefined
ProductView.js:27 sideOpen CHANGED TO true
```

The reason for the side panel getting stuck open is evident in the last 3 statements. First, notice the side panel was closed (`sideOpen CHANGED false`), which, as the next logged statement shows, triggered the `useEffect` that set `selectedProduct` to `undefined`. The change in `selectedProduct`, however, then triggered the other `useEffect`, which, after logging the final statement above, updated `sideOpen` back to `true`.

The fix is to include conditionals so that the setters are called only as necessary.

```
    // Open side panel when product is selected
    useEffect(() => {
        console.log(`selectedProduct CHANGED TO`,
selectedProduct);
        if (selectedProduct)
            setSideOpen(true);
    }, [selectedProduct]);

    // Deselect product when side panel is closed
    useEffect(() => {
        console.log(`sideOpen CHANGED TO`, sideOpen);
        if (!sideOpen)
            setSelectedProduct();
    }, [sideOpen]);
```

Congratulations on getting the bugs worked out of your implementation! A key skill for developers is trying every path a user could take to ensure the experiences are both correct and pleasing.

Re-rendering in components

Add a `console.log` before the return statement in `ProductView`. Refresh the application in the browser and click through several scenarios for opening and closing the panel and selecting items in the list. Notice how often the output displays in the console.
Add a `console.log` before the return statement in `ProductListItem` and/or `ProductDetails`. Refresh the application in the browser and click through several scenarios for opening and closing the panel and selecting items in the list. Notice how often the output displays in the console.

Bonus Phase: Remember state using local storage

There are times when it would benefit your users if the application remembered some of the settings on refresh or between sessions. The `useEffect` hook is one way (probably

the best way) to put the value of a state variable into `localStorage`. Likewise, the `useState` definition is often the best time to get the value back out.
Remember that there will be no value in `localStorage` the first time a user visits your application, so you'll need to check for that and provide an appropriate default. (You probably want the panel to be open on a user's first visit since it contains the welcome message.)

In this bonus phase, store the state of the side panel (whether opened or closed) in local storage. You can view (and edit or delete) local storage in the JavaScript tools in your browser by going to the "Application" tab and looking for "Local Storage" under "Storage" on the left.

*HINT: If you get stuck trying to figure out how to determine if the boolean `false` has been put into `localStorage`, note that `localStorage` stores and returns all values as strings and that **any** non-empty string (including "false") is considered **truthy**.*

Did you find this lesson helpful?

No

Yes