Exercise 05 – Props and State

Objectives

To be able to use props and state in components.

Overview

In this exercise you will add Props and State to the **FilterableProductTable**, displaying all the products in the array supplied. We will identify what data should be state and then home this in the appropriate component. Once we have state, we will deal with the user input to make our table reflect the search parameters chosen by the user.

Exercise Instructions

Check that the starter project runs correctly

1. In the command line, navigate into the **starter** folder for **EG05**.
2. Run the command:

```
npm start
```

3. Verify that the browser outputs the **Filterable Product Table** as at the end of the last exercises

Thinking in React Reminder:

1. Break the UI into a component hierarchy	*- DONE*
2. Build a static version in React	*- DONE (ALMOST)*
3. Identify the minimal (but complete) representation of UI state	- TODO
4. Identify where your state should live	- TODO
5. Add inverse data flow	- TODO

Complete the static version

To complete the static version, we need to populate the Product Table with the products and product categories as listed. For practice, we are going to create a context for Products, so that we can use it wherever we need to.

Provide products to ProductTable as props

1. In the **src** folder, open the file called **products**. You will see an exported **const** of **PRODUCTS** has been provided here - an array of product objects - ready for import elsewhere.
2. In **FilterableProductTable**, add an **import** for **PRODUCTS** from the **products** file.
3. Add a property called **products** to the **ProductTable** and set its value to **PRODUCTS**
4. Save the file and check the **React** tab in the developer tools, locating the **ProductTable** and verifying that it now has a **props** value which is the **products** array.

Use the products prop to populate the table

The first thing to do here is to pass **props** into the function. Once we have done this, we are going to perform a little bit of manipulation of this array so that we can extract the categories and the name and price of each product. As we do this, we will create an array of components that we will render in place of the current components.

1. In the **ProductTable** file, populate the arrow function parameter brackets with **props**.
2. Before the **return** in this function, declare a variable called **rows**, initialised as an *empty array*.
3. Declare another variable, this time called **lastCategory** and set this to be null.
4. Write a **forEach** loop with a *callback* that takes a product and:
   a) **IF product.category** does *not equal* **lastCategory**:
      i) Push a **ProductCategoryRow** with a *prop* of **category** set to **product.category** and **key** set to **product.category** to the **rows** array;
      ii) Set **lastCategory** to be **product.category;**
5. Push a **ProductRow** with a prop of **product** set to **product** and **key** set to **product.name**.
6. Replace the **<ProductCategoryRow />** and **<ProductRow />** with **{rows}**.

This essentially gets React to render the array created in the **forEach** loop.

7. Save the file and check the output.

You should see that the category *Electronics* and the *iPhone 6* product details are displayed **6** times. That is because we have not modified the **ProductCategoryRow** or **ProductRow** components to use their *props*.

8. Open the **ProductCategoryRow** component and add **props** to the parameters of the function.
9. In the **return**, *replace* the static text of **"Electronics"** with **{props.category}**.
10. Save the file and check that you now have the 2 different categories, even though the product details have not changed.
11. Open the **ProductRow** function and use the **props** in the function.
12. Set the **td** for the *Name* to be **props.product.name** and the **td** for *Price* to be **props.product.price**.
13. Save the file and you should find that we now have a static version as per the mock we were provided with.

Identify UI state

Remember to ask the following questions about state:

- Is it passed in from a parent via props?
- Does it remain unchanged over time?
- Can you compute it based on any other state or props in the component?

No actual coding is needed in this part but it would be good to make a note of what you decide.

Read on to find out what you should have decided…

- Original list of products – passed in as props – **not state**
- Search text user enters – **state**
- Value of checkbox – **state**
- Filtered list of products – can be computed by combining original list of objects, search text and checkbox value – **not state**

Identify where state should live

If you have successfully negotiated the last section, you should have concluded that states in the application are:

- The search text that a user enters
- The value of the 'in stock only' checkbox

The next stage is to decide where in the application this state should be held. Run through the following process to work out where you think state should be.

- Identify every component that renders something based on state
- Find common owner component
- Either common component or component even higher up should own state
- If no component makes sense, create new component to hold state and add it into the hierarchy above the common owner component

Both of these values are obtained from **SearchBar**, so it would be a reasonable conclusion to put the state there. However, as the filtered list of products is computed partially based on state, this is not the case. The **ProductTable** needs to receive information about the text in the search box and the status of the checkbox. The common ancestor of these components is **FilterableProductTable** so that's where we'll add state.

1. Open **FilterableProductTable** and add a **constructor** function. Be aware that constructor functions should receive **props** and call **super** with **props** too.
2. Under the **super** call, create an instance variable called **state** and set it to be an *object* that has:
   - a property **filterText** as an *empty string*;
   - a property **inStockOnly** as *false*.
3. Add a property of **searchDetails** to the **SearchBar** and **ProductTable** components in the render method, both set to **this.state**.

We are passing in the whole state object here as it is required in both components.

4. In **SearchBar** set:
   - The **value** of the *text input* to **this.props.searchDetails.filterText**;
   - The **checked** attribute of the *checkbox* to **this.props.searchDetails.inStockOnly**.
5. Save the file and check the browser.

Things seem temporarily broken, as we cannot type in the text input or change the checkbox. The section on adding inverse data flow will deal with that!

We still need to get the **ProductTable** to react to the state of the application. We are going to add some code that will execute before the creation of the array of elements happens for each product. We are going to see if the product contains the **filterText** or if the product meets the **inStockOnly** requirements.

6. Immediately after the *opening brace* for the **forEach** *callback*, add 2 variables:
   - Call the first one **lowerCaseProductName** and set it to the **product.name** *converted to lower case*;
   - Call the second **lowerCaseFilterText** and set this to **props.searchDetails.filterText** *converted to lower case*.

Doing this negates the issue of our user having to use the exact case of our product name.

7. Under this, add an **if** statement that checks to see if either of the following conditions are **true**:
   - **lowerCaseProductName** is *not* in **lowerCaseFilterText** (HINT: use indexOf and -1)
   - the *boolean* result of an **AND** between the inverse of **product.stocked** and **props.searchDetails.inStockOnly**.

   The if statement should simply **return** if either condition is met.
8. Save the file and check the browser - the original table is still present.

Add inverse data flow

To make the **SearchBar** work properly, we need to be able to update the state to reflect what the user has provided. To do this, we will raise a change event when the value of the input changes and pass this event to a function that will call another function in the **FilterableProductTable**. We will pass in the data to update state with and write functions to handle all of this.

Let's start with the handler functions in the **FilterableProductTable**.

1. Under the **state** declaration in the **constructor** declare another instance variable called **handleFilterTextChange** and set it to an *arrow function* that should:
   - Take an argument called **filterText**;
   - Call **setState** with **filterText**.
2. Add another instance variable called **handleInStockChange** and set this to an arrow function that should:
   - Take an argument called **inStockOnly**;
   - Call **setState** with **inStockOnly**.
3. Add **props** to the **SearchBar** that have the same names as the handler functions and call pass the handler function as a property.

```
<SearchBar
    …
    handleInStockOnlyChange={this.handleInStockOnlyChange}
    handleFilterTextChange={this.handleFilterTextChange}
    …
>
```

What we're doing here is creating function to receive the data from the **SearchBar** input and using **setState** to update the component tree's state. As both **SearchBar** and **ProductTable** receive **state** as **props**, any *changes will be reflected on re-render*.

To complete this section, all we need to do is tell **SearchBar** what to do with the new props we have just given it.

1. Add a **constructor** to **SearchBar**.
2. Within the **constructor** add an instance variable **handleFilterTextChange** and set this to be an *arrow function* that:
   - Recieves an **event** as an argument.
   - Calls **this.props.handleFilterTextChange** passing in **event.target.value**.
3. Add another instance variable **handleInStockOnlyChange** and set this to be an arrow function that:
   - Receives an **event** as an argument
   - Calls **this.props.handleInStockOnlyChange** passing in **event.target.checked**.
4. Add **onChange** attributes to the **inputs**, setting their value to be the *appropriate handler function*.
5. Save the file and check the browser.

Your **FilterableProductTable** should now be fully functioning!