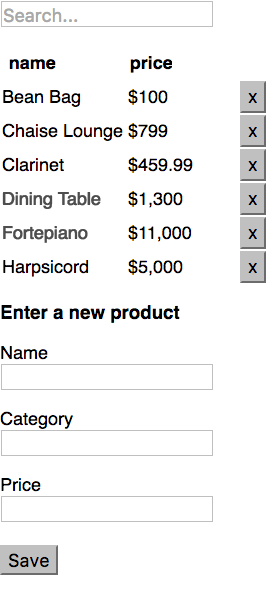
**CS648   
Assignment 12**

In this assignment, your job will be to work with React and Bootstrap to create a basic inventory management system. In general, this project will be broken down as follows:

* [Define the Requirements](#define)
* [Identify Components](#identify)
* [Build Out the Static Application](#build)
* [Identify State](#state)
* [Handle User Input](#userinput)
* [Style the Application with Bootstrap](#stylize)

**Define the Requirements**

Suppose we've been tasked with creating an application that renders a searchable list of products in an imaginary store and allows the user to add and delete products from the list. The project manager has provided you a mockup and it looks something like this:



What we see:

* There's a search input at the top that should filter the products as you type so that only products that match the current search string are shown.
* On the right of each row in the table, there are buttons to remove any of the items from the list.
* At the bottom, there's a form to create a new product. When you hit "Save", it should add the new product to the list.

We're not going to address how we get the data or save it after changes have been made. Instead assume that there is a JSON API that provides the data like this (replace the categories, prices, and product names with your own):

let PRODUCTS = {  
 '1': {id: 1, category: 'Music', price: '$459.99', name: 'Clarinet'},  
 '2': {id: 2, category: 'Music', price: '$5,000', name: 'Cello'},  
 '3': {id: 3, category: 'Music', price: '$4,500', name: 'Tuba'},  
 '4': {id: 4, category: 'Furniture', price: '$799', name: 'Chaise Lounge'},  
 '5': {id: 5, category: 'Furniture', price: '$1,300', name: 'Dining Table'},  
 '6': {id: 6, category: 'Furniture', price: '$100', name: 'Bean Bag'}  
};

Use create-react-app to begin the process of building out your application:

npx create-react-app assignment12

Once that’s done, you’re ready to identify the components and write a static version of the application.

**Identify Components**

You’ll want your components to be as simple and as easy to understand as possible. Each component should represent a discreet bit of logic and have a single purpose. Start by walking through the process of breaking up the mockup into component parts:

* a search field that queries the list
* a table of products, which is itself made up of:
  + rows of products, that each have a delete button
* a form for adding to the list, which is made up of:
  + three text inputs and a 'save' button

The products table seems like a component by itself, perhaps with a nested component for the delete buttons. Everything on this list pertains to products in some way, so it would be best to nest these components under a Products component that will be responsible for rendering everything. The component tree for the application ends up looking like this:

* Products
  + Filter
  + Product Form
  + Product Table
    - Product Row

**Build Out the Static Application**

Begin this assignment by creating a static version of the web page first, then add the interactive functionality. Begin by fleshing out your component outline with some boilerplate component files (placed within the ./src/ directory):

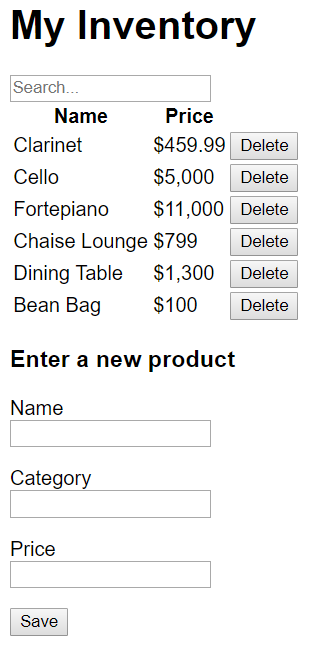
* **Product.js**  
  Imports Filters, ProductTable, and ProductForm components  
  Defines the data model (JSON data) \*  
  Renders a static heading and three components including <Filters>, <ProductTable>, and <ProductForm> in that order.
* **Filters.js**  
  Renders a form and a text box inside the form for search and filtering out our product list
* **ProductTable.js**  
  Imports ProductRow component  
  Maps our data model (represented as an object) to an array \*\*  
  Iterates through that array (.forEach()) and pushes a programmatically constructed <ProductRow> component to a temporary array. Call it ‘rows’ to makes things easy \*\*\*  
  Renders a table. Table should contain headings and render out product row (now represented by rows array)
* **ProductRow.js**  
  Renders a table row complete with <td> tags  
  Each <td> tag will display name and price (retrieved from props), and a button to delete an item
* **ProductForm.js**  
  Renders a form that displays text boxes for name, category, and price. Also renders a submit button for ‘Save’.
* **App.js** (already created for you)  
  Imports Products component  
  Renders the <Products> component

*\* Note: You’ll need to include your JSON data somewhere. Since the data will flow "down" to all the components that need it, it makes sense to represent this data in the highest level component. In this case, that would be the 'Products' component. To pass the data down to child components (<ProductTable>) you'll use the props attribute: <ProductTable* ***products={PRODUCTS}****></ProductsTable>*

*\*\* Note: To render a collection in React, you’ll need to build up an array of <ProductRow> components by iterating over props.products. Since the data model is an object, not an array, you'll need to map it to an array using the Object.keys() and .map() functions.*

*\*\*\* Note: Remember that JSX tags are compiled to normal JavaScript objects, and can be stored and passed around as such. This is pretty straightforward, but take note of the key attribute on the <ProductRow> component. Whenever you're building a list in React, you need to add unique keys to the items in the list, so that React can keep track of them when their order changes and avoid unnecessary re-rendering.*

At this point, you should have something similar to the image below:



You should be able to make changes to the data model and see them reflected in the UI. Try changing the number of products, or their attributes, and see how it's reflected in the UI.

**Identify State**

So far, the application has no state. There's essentially no user interaction and nothing ever changes. The task now is to identify what information can change as the user interacts with the application, and to represent that information in code as 'state'. Once you’re done with this part, the application still won't be interactive, but you'll have defined the bits of data that can change.

The most obvious element of state that you have beyond the form elements is the data model itself. You have buttons that can delete list items and a whole form that can create new ones. That means the list will have to be represented in state somewhere. Since the data is currently in the <Products> component, that might be a good place for state to live.

The remaining elements of state are the values in the form elements. You have three components with form elements that all affect the presentation of the products table. The search box (in the Filters component) will affect what is displayed. The product form and the delete buttons will affect the list directly as well.

Since the form and the filters live outside the product table component, the current list data and the currently applied optional filter will have to be stored in a shared component -- in this case, the Products component.

You've already located the list data there, so you just have to think about how to lift the state of the currently selected filters to the Products component! Here’s what you need to do in Products.js to get this done:

* **Products.js**  
  Add the constructor and super() to the Products class and pass in props  
  Create two properties in state: filterText = ‘’ and products = PRODUCTS  
  Pass the state of filterText down to the <ProductTable component: filterText={this.state.filterText}  
  Pass the state of the current list of products down to the <ProductTable> component: products={this.state.products}

<ProductTable> now has to take the list of products and optionally filter it according to the currently entered filter. This means it has to know:

* The state of the current list:
  + this should be passed in via props.products
* The state of the currently entered filter:
  + this should be passed in via props.filterText

Here’s what you need to do in ProductTable.js:

* **ProductTable.js**  
  Add a conditional within the .forEach() that checks to see if props.filterText is contained within product.name. Use the .indexOf() string method to handle this.

Refrain from looking at the rendered app now. It won’t look or function any different. In fact, the search form may not even work at all! Typing in the search field should render nothing. That's because we're controlling the value of the input element. In the next section you'll handle taking the user input.

**Handle User Input**

All that remains is for the user input to update state correctly in the application. This will be done with event handlers. Essentially, you're going to write a bunch of event handlers that just call setState. The complexity here lies in letting a child component's events affect its parent's state. The DOM's pattern for this type of communication is to bubble events up the DOM tree. By contrast, instead of explicitly bubbling actions up, you'll pass handlers down – just like any other kind of information, via props – that when called, will make the necessary changes to the parent's state.

To illustrate how to handle events by passing handlers down, start at the top of the component tree. The <Products> component's state is affected by three user actions that are triggered in descendent components:

* filtering the list of products
* submitting the new product form
* destroying products

Start with filtering the list of products. You simply want to set local state to match whatever data is entered in the <Filters> component. This involves:

**Product.js**

Writing a handler that takes input and passes it to setState:

handleFilter(filterInput) {  
 this.setState(filterInput)  
}

Binding the handler in the constructor function so it doesn't lose a reference to this when triggered on its child:

this.handleFilter = this.handleFilter.bind(this)

Passing the handler to the child component (on <Filters> for instance):

onFilter={this.handleFilter}

**Filter.js**

Now, in the Filters component, you will bind onChange on the input, and figure out a way to pass the value to the handler you’ve passed in. Just as in the parent component, this involves three things:

Writing an event handler that passes user input to the handler passed in from the parent:

handleChange(e) {  
 const value = e.target.value  
 const name = e.target.name  
  
 this.props.onFilter({  
 [name]: value  
 });  
}

Binding the handler in a constructor function so it doesn't lose a reference to this when triggered on the event target element:

constructor(props) {  
 super(props)  
 this.handleChange = this.handleChange.bind(this)  
}

Assigning the handler to the onChange property of the input element (You'll also assign a name property here, so we can reference it in the handler above):

... name="filterText" onChange={this.handleChange} ...

**Product.js (again)**

The other two actions that concern the <Products> component's state are those that add and remove products from the list. You'll start with adding, which happens via the new products form. The process will be much the same as before! Write a handler, binding, pass the handler to the component:

Write a handler that will take values from the form and use them to modify the state of the data model. Products in your list have unique IDs, so you'll generate one here. When working with data from a server, you would probably not generate an ID client side, and might use some other unique key instead. Note the callback form of the arguments to setState here. We use the callback because in order to replace the list with a new version of itself, we need a reference to the existing state of the list. Whenever you are using existing state or props to calculate the new state, you need to use this callback form of setState:

handleSave(product) {  
 if (!product.id) {  
 product.id = new Date().getTime()  
 }  
 this.setState((prevState) => {  
 let products = prevState.products  
 products[product.id] = product  
 return { products }  
 });  
}

Bind that handler in the constructor function so it doesn't lose its reference to this when passed to the child component:

this.handleSave = this.handleSave.bind(this)

Pass the handler to the <ProductForm> component so it can use it when submit is triggered on the form:

<ProductForm onSave={this.handleSave}></ProductForm>

**ProductForm.js**

Handling the form submission works just like the process above. Write a handler, binding, pass the handler to the component. Starting to see a pattern?

Write a handler for the submit button:

handleSave(e) {  
 this.props.onSave(this.state.product)  
 this.setState({  
 product: Object.assign({}, RESET\_VALUES), errors: {}

})  
 e.preventDefault(); //Prevent form from triggering HTTP POST  
}

Bind it in a constructor function:

constructor(props) {  
 super(props)  
 this.handleSave = this.handleSave.bind(this)  
}

Pass it to the button elements onClick hook.

onClick={this.handleSave}

Notice the RESET\_VALUES constant in the setState call. Above your class, create this constant so that we make sure we can clear the values once the user submits the form:

const RESET\_VALUES = {id: '', category: '', price: '', name: ''}

You’ll also need to account for this in the constructor:

this.state = {  
 product: Object.assign({}, RESET\_VALUES), errors: {}  
}

So far, so good, but remember that your form inputs don't work right now because you've bound their values to local state in your component. You've got to complete the work to make the form inputs update state on user input. Once again:

Write a handler. Again, you're using previous state to calculate the new state, so you'll have to use the callback form of setState arguments:

handleChange(e) {  
 const target = e.target  
 const value = target.value  
 const name = target.name  
  
 this.setState((prevState) => {  
 prevState.product[name] = value  
 return { product: prevState.product }  
 })  
}

Bind the handler in the constructor:

this.handleChange = this.handleChange.bind(this)

Attach them to the inputs via handler properties:

... onChange={this.handleChange} value={this.state.product.name} ...  
... onChange={this.handleChange} value={this.state.product.category} ...  
... onChange={this.handleChange} value={this.state.product.price} ...

The final event to be concerned about is the delete buttons on each product row. As you've defined within your components, the product row is actually two steps below the <Products> component. No worries, you just have to pass your props an extra time. In <Products>, create the handler and pass it as before:

**Products.js**

Write the handler:

handleDestroy(productId) {  
 this.setState((prevState) => {  
 let products = prevState.products  
 delete products[productId]  
 return { products }  
 })  
}

Bind it in the constructor:

this.handleDestroy = this.handleDestroy.bind(this)

Pass it to the <ProductForm> component via an onDestroy hook:

onDestroy={this.handleDestroy}

**ProductTable.js**

Write the handler:

handleDestroy(id) {  
 this.props.onDestroy(id)  
}

Bind it in the constructor:

constructor(props) {  
 super(props)  
 this.handleDestroy = this.handleDestroy.bind(this)  
}

Instead of passing the handler to an input's event handler hook, we pass again to a second child component:

rows.push(  
 <ProductRow product={product} key={product.id} **onDestroy={this.handleDestroy}** />  
)

And finally, to complete the loop, you'll repeat this in the <ProductRow> component:

**ProductRow.js**

Write the handler:

destroy() {  
 this.props.onDestroy(this.props.product.id);  
}

Bind it in the constructor:

constructor(props) {  
 super(props)  
 this.destroy = this.destroy.bind(this)  
}

Pass it to the button via an onClick hook:

onClick={this.destroy}

The whole step of adding interactivity turns out to be extremely regular: just keep writing handlers, binding them, and passing them down. The key is having already mapped out state in the application. If you're working on a React application and having trouble handling user input, most likely there is something wrong with how you've located state in your application. Sometimes you might have the same piece of state tracked in two different components. The fix should involve storing each piece of state in just one place, and having that be at the highest point in the component tree where that state is referenced.

Stylize Application with Bootstrap

The last step in this assignment is to stylize your application using Bootstrap. The final product should look similar to the design outlined below:

