MernStack Project Notes – Udemy Course  
React Tips  
  
**Tip**: Working with react router dom  
**Tip**: props.children in react  
**Tip**: React.Fragment  
**Tip**: Conditionally render content  
**Tip**: Portals  
**Tip**: Setting up google maps for your project  
**Tip**: Props destructuring  
**Tip**: Redux and react hook – (useState and useReducer)  
**Tip**: Breaking the project down  
**Tip**: useEffect  
**Tip**: Wiring up a custom confirm (And passing in two component as one using React.Fragment)  
**Tip**: How he setup form validation using a custom hook and re-usable components  
**Tip**: How to use the useContext Hook for security in our app  
**Tip**: Using a model View design in our app for our express.js routes (**another way of exporting modules as well**)

**Tip**: useEffect  
useEffect allows you to call side effects in pure functions. It allows you to perform an action at every re-render or state change.  
If you use useEffect without a dependent variable, it will run after ever render. If you supply a dependent array, it will only run when a value in the array changes.  
<https://www.w3schools.com/react/react_useeffect.asp>  
Graphical user interface, text, application, email

Description automatically generated

**Tip**: Breaking the project down  
A screenshot of a computer

Description automatically generated with medium confidence  
  
It’s a pretty straightforward project  
There are “2” main parts to the application  
He has the places and user components for the base UI  
He has a shared folder for all of the supporting and re-usable elements  
The way that he has the places and user components:  
User.js  
  
Graphical user interface, application

Description automatically generated

In a nutshell:  
He has a “PAGES” folder for the base Parent Component (Base UI)  
The Data (test data for starters is behind the main Page Component)  
He has the component’s folder for all of the supporting components that are children of the base page(s) component(s).  
Main Page Component for **User**: Users.js  
A picture containing text

Description automatically generated  
Main Page Component for **Places**: UserPaces.js  
Graphical user interface, application, PowerPoint

Description automatically generated  
The components for buttons and input fields are in the Shared Components folder  
Graphical user interface, text, application

Description automatically generated  
  
  
  
Text

Description automatically generated  
  
He splits the implantation by using parent and child compositions  
The props are passed down from parent-to-child  
**Users Is three Levels deep**  
**Users.js** – Parent Component – has the data const USERS = []  
  
Text

Description automatically generated  
  
The child component – **UserList.js** – he passes the USERS[] data to it  
Text

Description automatically generated  
  
The props are passed down to the third component (**UseItem.js**) (child of UserList)  
Text

Description automatically generated

Text

Description automatically generated  
  
End Result:

Graphical user interface, website

Description automatically generated

For places:  
Graphical user interface, text, application

Description automatically generated  
  
He uses the same concept, he uses custom re-usable buttons and inputs as child components  
In the pages folder, this is generally the base view component  
Text

Description automatically generated  
  
The way he wired up the <Input /> component  
Based on our tutorial / explanation of how useReducer works, he implemented the component to wire up validation with it. This is achieved easily with useReducer (when you want to add complex logic) in your component level state management.

A screenshot of a computer

Description automatically generated with medium confidence  
  
It's actually kinda slick the way he did it Afterall (I initially did not like the way he did it, but after studying it, I actually understand why he did it that way). When writes state with the reducer function, he also adds a custom validator to validate the input.  
  
He also created a formValidator as well with a custom hook to determine whether the form’s input was valid or not. (a little bit of an overkill)  
Look at Lesson 62 on how he initially wired up the form validator – As shown below, he is using the form validator to determine whether there is input in all fields and all field’s data is valid before he enables the Add Place button (see image below)  
Graphical user interface, text, application

Description automatically generated

**Tip**: Redux and react hook – (useState and useReducer)  
**What is a redcuer**?  
The reducer is nothing else but a **simple function** that accepts **two arguments** and **based on those two arguments, returns a new state value.**

**Javascript Reduce**  
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce>  
The reduce() method executes a **user-supplied "reducer" callback function on each element of the array**, in order, passing in the return value from the calculation on the preceding element. The final result of running the reducer across all elements of the array is a single value.  
  
Text

Description automatically generated

const array1 = [1, 2, 3, 4];

const initialValue = 0

const newSum = array1.reduce((prevVal,curVal) => prevVal + curVal,initialValue);

console.log(newSum) //10  
  
Works a lot like the JavaScript **.map function  
  
Another example**const reducer = (accumulator, currentValue) => accumulator + currentValue;  
[2, 4, 6, 8].reduce(reducer)  
//20

**Reflecting back on what we explained earlier about what a reducer is an what javascript reduce is:**In React, useReducer essentially accepts a reducer function that returns a single value:

const [count, dispatch] = useReducer(**reducer**, initialState);

The reducer function itself accepts two parameters and returns one value

The first parameter is the **current state**, and the **second is the action**. The state is the data we are manipulating. The **reducer function receives an action**, which is executed by a dispatch function:

function reducer(state, action) { }

dispatch({ type: 'increment' })

The **action is like an instruction you pass to the reducer function**. **Based on the specified action**, the reducer function **executes the necessary state update.**

<https://blog.logrocket.com/react-usereducer-hook-ultimate-guide/#:~:text=type%20is%20dispatched.-,useState%20vs.,you%20can%20do%20with%20useState%20>.

The golden rule still remains. Component state for component state, Redux for application state.  
  
useReducer:  
How does the useReducer Hook work?

The **useReducer** Hook is **used to store and update states**, just like the **useState** Hook. It accepts a **reducer function** as its first parameter and the **initial state** as the second.

useReducer returns an array that holds the current state value and a dispatch function to which you can pass an action and later invoke it. While this is similar to the pattern Redux uses, there are a few differences.  
  
For example**, the useReducer function is tightly coupled to a “specific reducer**.” We dispatch action objects to **that reducer only**, whereas in Redux, the dispatch function sends the action object to the store. At the time of dispatch, the components don’t need to know which reducer will process the action.

**Putting it all together (I created a component in my project)  
Text

Description automatically generated  
  
Graphical user interface, text, website

Description automatically generated**

useState vs. useReducer

useState is a basic Hook for managing simple state transformation, and useReducer is an additional Hook for managing more complex state logic. However, it’s worth noting that useState uses useReducer internally, implying that you could use useReducer for everything you can do with useState.

**When to use useReducer**useReducer provides more predictable state transitions than useState, which becomes more important when state changes become so complex that you want to have one place to manage state, like the render function.

A good rule of thumb is that when you move past managing primitive data, i.e., a string, integer, or Boolean, and instead must manage a complex object, like with arrays and additional primitives, you’re likely better off with useReducer.  
  
From  
[**https://blog.logrocket.com/react-usereducer-hook-ultimate-guide/#:~:text=type%20is%20dispatched.-,useState%20vs.,you%20can%20do%20with%20useState%20**](https://blog.logrocket.com/react-usereducer-hook-ultimate-guide/#:~:text=type%20is%20dispatched.-,useState%20vs.,you%20can%20do%20with%20useState%20)**.**

**Excellent example below**Text

Description automatically generatedGraphical user interface, text, application

Description automatically generatedNotice how we are dealing with all these state transitions, like username, password, isLoading, error, and isLoggedIn, when we really should be more focused on the action that the user wants to take on the login component.

**We used five useState Hooks**, ***and we had to worry about when each of these states is transitioned***.

We can refactor the code above to use useReducer and encapsulate all our logic and state transitions in one reducer function:  
  
Graphical user interface, text, application

Description automatically generated  
  
Graphical user interface, text, application

Description automatically generated  
  
Text

Description automatically generated  
  
Notice how the new implementation with useReducer has made us more focused **on the action the user** is going to take.  
  
After reading this article I can see why useReducer actually is more cleaner than handing a bunch of useState transitions.

**Tip**: Working with react router dom  
npm i react-router-dom  
  
Tips Below on Routing:  
The path “/” means default route  
The exact means that the route needs to be exact  
The <Redirect to “/” means that if a route is entered that does not exist, use the default route  
The <Switch> means that since routes are executed in order, it acts like a switch statement in code to stop at the entered route

Text

Description automatically generated

**Tip**: props.children in react  
Below is a perfect example of props.children  
Props.children takes all of the content passed in (**html elements etc..)** and you reference  
all of the content via props.children (AHHHHHHHH….)  
Text

Description automatically generated  
  
A screenshot of a computer

Description automatically generated

**Tip**: React Fragment  
React Fragment allows you have more than one root element. Javascript has a limitation of only allow for one root element when rendering, so in order to get around it, you have to wrap two root elements in a React.Fragment  
  
Text

Description automatically generated

**Tip**: Conditionally render content  
Component: MainNavigation.js  
Method One – Using ternary way:  
Text

Description automatically generated  
  
Method 2: (Preferred shortcut way)   
Text

Description automatically generated  
– I used this method also in my Nutrition Service Application as well  
  
Text

Description automatically generated  
  
  
**Tip**: Portals  
<https://reactjs.org/docs/portals.html>  
Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.  
In Your index.html file:  
Text

Description automatically generated  
  
We can create a hovered component for navigation

Text

Description automatically generated  
  
<https://www.w3schools.com/tags/tag_aside.asp>  
The <aside> tag defines some content aside from the content it is placed in. The aside content should be indirectly related to the surrounding content. Tip: The <aside> content is often placed as a sidebar in a document.  
  
Then in your navigation  
Text

Description automatically generated  
  
You can toggle on and off

Graphical user interface, application

Description automatically generated  
  
A screenshot of a video game

Description automatically generated with medium confidence

**Tip**: Setting up google maps for your project  
First go over to Google Maps Platform  
Since google charges, I will use the free mapbox

https://account.mapbox.com/auth/signup/  
Graphical user interface, application

Description automatically generated  
  
pwd= bflu@ (Not going to implement maps) because it still requires a credit card  
<https://www.udemy.com/course/react-nodejs-express-mongodb-the-mern-fullstack-guide/learn/lecture/16855052#overview>

**Tips**: Props destructuring  
Destructuring was introduced in ES6. It’s a JavaScript feature that allows us to extract multiple pieces of data from an array or object and assign them to their own variables.  
  
<https://medium.com/@lcriswell/destructuring-props-in-react-b1c295005ce0>  
  
Graphical user interface, application

Description automatically generated  
  
So in our project:  
A screenshot of a computer

Description automatically generated with medium confidence  
  
Instead of using **props**.id or **props**.onInput, we can just use destructuring to make it cleaner and shorter and just use onInput and id

**Tip**: Wiring up a custom confirm (And passing in two component as one using React.Fragment)

Graphical user interface, text, application

Description automatically generated  
  
Component: PlaceItem.js & Modal.js  
Notice how we are passing in single prop for footer that consists of two components. You can pass in two components this way by way of React.Fragment  
Text

Description automatically generated  
He also is using React.Portal to give the Modal a Z index outward render. (Like a popup)  
  
**Tip**: How he setup form validation using a custom hook and re-usable components  
The way this was setup is a little confusing but explained below:  
He uses custom input controls, the file:  
import Input from '../../shared/components/FormElements/Input'  
Text

Description automatically generated  
  
Input.js is the custom input component  
Text

Description automatically generated  
  
He passes his props for the custom input control the normal way. Any change made to any data in any controls receives events through his custom input control, input.js  
The Input.js file receives the validator type as a parameter, the data to validate from the data passed in as props (e.target.value)  
When a handler is fired off, it dispatches an action to the input reducer in the input.js file, this reducer calls the validate action from the Validators file  
  
  
  
  
  
Text

Description automatically generated  
  
Based on the type specified, it validates the data against his logic to determine whether the data is valid or not and returns a isValid flag  
  
To further validate the form, there is a callback  
Text

Description automatically generated  
Whenever there is a state change, the useEffect method calls this via a custom hook

A screenshot of a computer

Description automatically generated with medium confidence  
  
When the change is made in state, it triggers logic in the custom hook’s form reducer that determine’s whether the form is valid based on input in the input field (the isValid flag that is set in the input reducer).  
It's a lot to wire-up and understand, but once you break it down, it’s not that difficult to grasp. Still kind of an overkill.

**Tip**: How to use the useContext Hook for security in our app  
<https://reactjs.org/docs/hooks-reference.html#usecontext>  
React Context is a way to manage state globally.

It can be used together with the useState Hook to **share state between deeply nested components more easily than with useState alone.**  
Wireup for security in my app  
First we create the context component

A screenshot of a computer

Description automatically generated with medium confidence  
  
Above we recreate anonymous functions as pointers to functions we want to include in our context  
Then in our app.js, we wrap the Context object around all of our components

Text

Description automatically generated  
  
We set the anonymous functions up.  
Then to consume the context object  
In our NavLinks

Text

Description automatically generated  
Graphical user interface, application

Description automatically generated  
  
  
In our user, auth login page  
Text

Description automatically generated  
  
Graphical user interface, application

Description automatically generated  
  
**Tip**: Using a model View design in our app for our express.js routes  
Graphical user interface, text, application

Description automatically generated  
  
We create a controllers folder and add the following code:

Text

Description automatically generated  
  
Notice how we export the file (we are not using the module.exports way)  
  
Then to use the file in our routes:  
Text

Description automatically generated