Unit 2 Sprint 3

**React Router**

Routing

* It is the way we navigate through websites and web applications
* Click on a link on any web app or website and you are routing to a URL and requesting some information that lives somewhere else

Server

* The term “server” refers to centralized resources on a network. These servers are physical devices that run the “behind the scenes” work of the internet like data storage

Diagram

Description automatically generated

Server-Side Routing

* When we request information from a server (by clicking a link or a button), that server then sends back the document that was requested

Pros

* A server-side route will only request the data that’s needed. No more, no less.
* Because server-side routing has been the standard for a long time, search engines are optimized for webpages that come from the server.

Cons

* Every request results in a full-page refresh. That means that unnecessary data is being requested. A header and a footer of a webpage often stays the same. This isn’t something you would want to request from the server again.
* It can take a while for the page to be rendered. However, this is only the case when the document to be rendered is very large or when you have slow internet speed.

Client-Side Routing

* When routing is handled internally by the JavaScript that is already on the page, we achieve what is known as Client-side routing. This tool is how we get things done, and it does so without refreshing the page
* When a user clicks on a requested resource, instead of the client asking for that resource from the server via a URL, JavaScript will prevent this

Pros

* Because less data is processed, routing between views is generally faster.
* Smooth transitions and animations between views are easier to implement.

Cons

* The whole website or web-application needs to be loaded on the first request. That’s why the initial loading time usually takes longer.
* Because the whole website or web-application is loaded initially, there is a possibility that there is data downloaded for views you won’t even come across.
* It requires more setup work or even a library. Because server-side is the standard, extra code must be written to make client-side routing possible.
* Search engine crawling is less optimised. Google is making good progress on crawling single-paged-apps, but it isn’t nearly as efficient as server-side routed websites.

A picture containing text

Description automatically generated

**React Router** - for rendering components based on changes to the path without making additional network requests

* Single page applications using Client-side networking is like a desktop application. Once it’s been downloaded all of the changes come from the application and no network requests are required

**Create React App**

* In terminal, type: npx create-react-app basic-routing
* In new terminal window, type: npm i react-router-dom
* In App.js cleanup unnecessary JSX and the React App import
* import { BrowserRouter, Route, Link } from ‘react-router-dom’;

Text

Description automatically generated

* + You can remove BrowserRouter from the import and the JSX in your app.js if you wrap <App /> in the index.js file with <BrowserRouter> instead
  + The <Route /> component declares which components will be mounted based on what URLs the user requests
    - Route components takes in a few props: 1, the URL path where the Route component will trigger 2, the component prop which will mount when the URL matches the requested path
    - Basically the Route components asks you this question every single time you set one up: what **component** do you want **mounted** when a user asks for what **URL path**?
    - By placing the exact prop on a <Route /> component you are saying that the specific path will only trigger if it matches the path requested exactly

<Route exact path="/" component={Home}/>

* The Link component will produce an anchor tag that will link to a pre-defined component of your choice

import { Link } from 'react-router-dom'

<Link to="/about">About</Link>

* The Switch statement is used when a route matches multiple routes, the browser will only render the first component it comes across.
  + Wrapping in Switch will allow us to remove the “exact” word and put least specific routes at the bottom and the more specific at the top:

Text

Description automatically generated

* In depth on **Route**
  + In React dev tools you can see more info on the props injected into a particular component:
    - Location: is all about where we are
    - Match: is all about how we matched the current route to where we are
    - History: contains actions we can do to navigate imperatively
      * 
  + Two useful props for Route:

1. Render gives us access to Location, Match, and History
   1. 
2. Children makes it so that the component supplied renders regardless of whether the path is a match to the current location or not
   1. 

* In depth on **Link**
* The **to** prop can take several different shapes, most simply a string containing the path name, query string, and hash
  + 
* The **to** prop can also take the shape of an object, which allows us to specify the different pieces separately (path name, query string, hash) More importantly it also allows us to specify some state
  + Text

    Description automatically generated
    - It has state so that the component that it matches has some knowledge of where we navigated from
      * 
      * A picture containing logo

        Description automatically generated
* **useParam** hook relies on props to pass new and changing data into the app. It allows us to create dynamic routes that will render content based on the URL

import { useParams } from "react-router-dom";

<Route path='/:handle' component={Profile} />

* **useRouteMatch** hook lets you add nested routes to your application such that you can change a single parameter in the URL without having to change the whole thing

import { useRouteMatch } from 'react-router-dom'

function App() {

const {path, url} = useRouteMatch();

return (

<li>

<Link to = {`${url}/employee-details`></Link>

</li>

<Switch>

<Route path = {`${path}/employee-details`}>

<employee-details/>

</Route>

</Switch>

)

}

* **useHistory** hook allows us to create dynamic elements (buttons, text, etc.) whose function changes based on a user’s history

import { useHistory } from 'react-router-dom'

function BackButton({ children }) {

let history = useHistory()

return (

<button type="button" onClick={() => history.goBack()}>

{children}

</button>

)

}

* Dynamic Routes



* + Inside of props.match.params we can get { id } corresponding to :id
  + A URL can have as many parameters as we wish
    - * Graphical user interface, text, application, chat or text message

        Description automatically generatedGraphical user interface, text, application

        Description automatically generated

Text

Description automatically generated

More realistically we would use the id to fetch the user from an API. The example below shows what that would look like using a fake endpoint called fetchUser:



Text

Description automatically generated

A screen shot of a computer

Description automatically generated with low confidence

**Form Management**

<form> element represents a document section that contains interactive controls for submitting information to a web server

<input> element is used to create interactive controls for web-based forms in order to accept data from the user

<label> element represents a caption for an item in a user interface

Two ways to use the <label> element:

1.Wrap the whole input with the element 2. Use **htmlFor** to id of the input

Text

Description automatically generated Text

Description automatically generated

**Creating a dropdown:**

Dropdowns require the use of the <select> element. Inside of that we use the <option> element which needs to have a value which is saved in the DOM

Text

Description automatically generated

**Creating a checkbox:**



**Creating a submit:**

****

On submit a query string (containing key value pairs, separated by &) is attached to the URL containing everything from the form



**onChange**

**Getting values from the DOM:**

* Inputs have a value property on them which can be evaluated by adding .value to them

favVehicle.value will be trucks if that is what was selected from dropdown

* Instead of value for a checkmark, however, you need to use .checked

**Keep current values inside of state:**

* Attach an event handler on an input in order to capture the typing event which stores the new value from the input. Now any area of the app can have knowledge of the value (In the example below it is the first name)

****

Text

Description automatically generated

* Adding || ‘unknown’ is helpful for when the user doesn’t enter anything



**onSubmit**

* This sends the information from the form to the backend
* When the user clicks submit the default of onSubmit on forms is to reload the page
* Need to add onSubmit to the form and then define it
  1. Use preventDefault() to keep the page from reloading
  2. Can use axios or post request (which would need a url and a payload)



Text

Description automatically generated

* Can make this look nicer by pulling the callbacks out instead of having them inline with the elements

Text

Description automatically generated

**Handling multiple inputs:**

Text

Description automatically generated Text

Description automatically generated

^ Before ^ ^ After ^

**Spread Operator example:**

Graphical user interface, text, application, email

Description automatically generated

**Computed properties**

let myObject = { firstProperty: "Hi Lambda!" };

Property can be accessed by using dot notation or bracket notation

myObject.firstProperty; //"Hi Lambda!"

myObject["firstProperty"]; //"Hi Lambda!"

* Under the hood, all object properties are strings. Dot notation is easier to type but sometimes it’s not legal to use it (when our property has a number)

let myOtherObject = {"3": "totally legal key/value pair"};

myObject.3; // JavaScript freaks out at you.

myObject."3"; // JavaScript freaks out at you.

myObject[3]; // JavaScript is pleased. It will implicitly coerce this integer to a string.

* If you don’t include quotes, JavaScript will attempt to evaluate what you wrote as a variable enabling us to evaluate an expression

let myThirdObject = { 1: "sup", 2: "hey" };

let computedProperty = 1;

myThirdObject[computedProperty]; // "sup";

myThirdObject[1 + 1]; //"hey";

**Managing form data with controlled inputs**

Text

Description automatically generated****

Live Lesson Day 2

Steps for form creation:

1. Set state (keep track of resources second slice is keeping track of form values)
2. Inside jsx we have a form containing two inputs and a submit button. Each input has at least four props but can add more (ie placeholder and maxLength)
   1. Text

      Description automatically generated
   2. Main form values: name, type, value, onChange
3. Edit const change to set what happens when user types
   1. Text

      Description automatically generated
   2. Note that [name] isn’t an array literal, it will override whatever the value of name was previously in formValues
4. Edit const submit
   1. Text

      Description automatically generated
   2. Rhiannon says to Google concat

**Advanced Form Management**

**Control forms and form inputs in custom forms**

Steps

(once you have created your form fields)

* 1. Add names to each input
  2. Use a single slice of state to keep track of all values in the entire form
  3. Add values to each input; dropdowns will also need hard coded values on option
  4. Add checked to each radio button
  5. Add onChange to every field
  6. Declare the onChange and define the variable you are setting it to
  7. Set up the event handler for that variable to include all of the details you need from the event.target so that when anything on the form changes, the event fires and the callback executes and stores all of the information we need
  8. Create a variable that sets the type to either a checkbox or the value (ternary)
  9. Set the value of the entire form, spreading out the current form and then adding to it

Text

Description automatically generated

**Validate user input in forms**

Steps

(once you have created control forms and form inputs in custom forms)

1. Add a submit button that uses state; starts out disabled
2. Set state with the state starting as true
3. Install a library that will allow us to build a schema that will allow us to validate shape of the state of the form
   1. Install npm i yup ~ import to top of page (import \* as yup from ‘yup’;)
4. Build out the schema form shape
5. When the state of the form changes we will fire an effect that will check the schema; useEffect
6. Create a slice of state to keep track of errors to show the user what needs to be fixed
7. Set a function to check to see if the value entered is in line with the shape and if not it will send the user those errors; include name of field being changed and the value to use; useEffect
8. Along with creating the function you also need to declare that function with the name and value
9. This declaration will also need to be set for validation errors; yup.reach
10. Use .then in the event that the validation passes
11. Use .catch in the event that the validation fails; err.errors[0]
12. Need to add to the change event
13. Need to create something to show the errors on the screen by using JSX

Text

Description automatically generated

**POST requests to pass data collected from a form to a database**

* It is a method that allows us to “post” (or create) information on a web server
* When a user makes a POST request, they are adding data to the server’s database
* Still need npm install axios
* The promise created by axios will resolve into a successful response or reject with an error
* .then() and .catch() may include:

1. Setting data into state in your component
2. Alerting the user to an error
3. Use the new data to create side effects in your component(s) that modify the interface for your user in some way

Steps

(once you have created control forms and form inputs in custom forms and validated)

1. Declare the onSubmit and define the variable you are setting it to
2. Set up the event handler for that variable to include event.preventDefault()
3. Install and import axios for using either axios or post
4. Can use axios or post request (which would need a url and a payload)
5. Create the payload (newUser) by pulling the details from the form
6. Use .trim() for the newUser and pull each detail from the form (ex: form.star)
7. Return promise; .then, .catch
8. You can now use the res.data for whatever you need it for (example just resets the form)

Text

Description automatically generated

**Andy’s Basic Form:**

1. start small
2. try to memorize how to do a simple form
3. it needs
4. some of state
5. some initial form values
6. a form with at least 4 things
7. `type` `onChange` `name` `value`
8. a button
9. a submit handler on the form
10. need a `change` handler
11. updates our state with the new values
12. we need a `submit` handler
13. creates a new form object to submit
14. submits it
15. resets our form