**ReactJS Notes**

**What is React js**

* a javascript library for building user interfaces
* client-side (runs on the user browser)
* the page doesn’t need to refresh as it does not need to request a new html page, making it super fast
* reduces code needed
* uses components which has dedicated, but small task
* we use react mainly for single page applications. Sometimes when we click a new tab, it looks like we requested a new html page, but we just changed what is visible
* there is html code in js code, which browser cannot understand which is why there is a ‘build’ process which makes the code readable from the browser, so the code we write is not exactly the same as the code the browser reads. Babel helps to convert JSX to plain JS.

**Installing React**

* first we need to install node js <https://nodejs.org/en/> and download the latest version
* check nodejs is installed by typing node -v into the cmd
* open cmd and cd into a folder where you want the react app to be (such as desktop)
* run in cmd: npx create-react-app INSERT\_APP\_NAME
* once it says happy hacking, you are done
* run in cmd: cd INSERT\_APP\_NAME
* run in cmd: npm start (as long as you keep this up, the website will update depending on code changes)
* this should open a tab on your browser at <http://localhost:3000/>

A picture containing logo

Description automatically generated

* open the react app in a code editor
* Text

  Description automatically generated
* Open src folder, and delete app.test.js, logo.svg, reportWebVitals.js, setupTests.js, and App.css
* You should be left with the following: Graphical user interface, application

  Description automatically generated
* In src, go to index.js and make it look like the following

Text

Description automatically generated

* In src, go to index.css and remove all of it

Graphical user interface, application, Teams

Description automatically generated

* In src, go to app.js and make it the following

Graphical user interface, application

Description automatically generated

* Open in vscode, go to view, extensions, and type prettier
* Graphical user interface, text

  Description automatically generated
* Install it, then go to file, preferences, keyboard shortcuts, type format document, and edit the first one to make a shortcut u want

**Diving into the project**

* We have two js files, index and app. Index is the starting point of our react application, the code in here will be the first code that will be executed in our browser.
* Notice we imported the ReactDOM object from the ‘react-dom’ library
* On the ReactDOM object, we call a render method which allows us to add html code. This html code works because of the build step. This html in js code is called JSX. JSX is not understood by browsers and is converted behind the scenes.



* Notice this code renders our own html element, the <App/> element which is imported from the App file.
* We tell the App element to be placed where the element of id of root is. (you can find the root in public/index.html)
* Index.html is the single html page in the project since react is mainly used for single paged applications
* Notice if you inspect the page, not ctrl U, and open the div with id root, we see code added there even tho we didn’t add it directly on the index.html page.

Text

Description automatically generated with medium confidence

**Components**

* Looking at App.js, App is a react component, which is just a standard js function, but it returns JSX code.
* Text

  Description automatically generated with medium confidence
* We export the function using ‘export default AppNAME;’ to make it accessible outside the file
* Also, notice when we inspect the page, we don’t see a div called App, we just see the return value of the App function. This is because these custom elements like App is not understood by the browser.
* We can change the app js code to look like the following
* We just write the html code we want to have appear on the screen

Text

Description automatically generated

* Notice the browser auto reloads to detect the changes
* Go to <https://github.com/academind/react-complete-guide-code/blob/zz-reactjs-summary/extra-files/index.css> and copy the css code and put it into index.css
* Go back to app.js and add the card class to a div using the className. Note we don’t use class like html since class is a keyword in javascript.
* Text

  Description automatically generated

**Building and Reusing Components**

* However, right now, all our code is in one component, and we might want to split that up in the future. For example, if we want a second todo, we have to replicate the entire code block. As well, if we want to change all the code blocks, you have to go to each of the code blocks and edit it manually instead of making the change in one location. That’s why react has components
* Create a new folder in the src folder called components
* Create a new file in the components folder which will be the name of your app. The name has to start with a capital to differentiate it from built-in html components.

Graphical user interface, application

Description automatically generated

* Now make app.js and todo.js look like the following
* Note the Todo app is self closing, we could also write <Todo></Todo>, but since there is not content between it, we can self close it.

Text

Description automatically generatedText

Description automatically generated

* If we want multiple Todo, we just have to Todo multiple Todo elements and one change in the Todo.js file will change all the Todo elements.

**Props and Dynamic Content in Components**

* We want to pass arguments into the components
* In app.js, add the following

Text

Description automatically generated

* In the component, we can add a parameter called props(we can name it whatever we want, but the idea is called props so we’ll use props)
* Props is a js object and all the arguments are key value pairs. So the attribute names such as ‘text’ would be keys will the value would be something like ‘Learn React’
* Text

  Description automatically generated
* Notice we use {} around props.text to tell react this is js and not html, since otherwise, the h2 name would be props.text. Inside the {}, we can have things like 2+2 and it’ll evaluate to 4. You can have any single line expression inside the {}, but you can’t have block expressions like if statements.

Graphical user interface, application

Description automatically generated

**Handling Events**

* If we want to listen to a button, we add an extra attribute
* Note: all the default HTML elements in the JSX are actually react components, allowing us to add certain attributes that we normally wouldn’t be able to with just HTML
* For any element, we can add an onClick attribute and let it equal some function



* We do not put () after the function name as that executes the function, we only want to point at it.

Text

Description automatically generated

**Adding More Components**

* Make a Backdrop and Modal.js file

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

Text

Description automatically generated

* Notice in Backdrop, we can make the div self-closing since it doesn’t have any content inside it.
* However, when we run the program, the website never changes, thus we need state

**State**

* In order to change what we see, we have to change the state of components
* React has a built in concept called state
* We need to import the useState function from ‘react’. We don’t need to have a path since it’s a third party library, we can reference the library name
* 
* With usestate, we register different states
* useState always returns an array with two elements which we can store within a constant with array de-structuring
* we can think of state as a variable that is managed by react
* in our case, we define the initial value to be false. We can access the currently stored value through the modalIsOpen and we change the value through setModalIsOpen function
* 
* Whenever we change the state via the setModalIsOpen, react will re-execute the component function to which the state belongs and update what is rendered on the screen

Text

Description automatically generated

* Notice when we click the button, it references the deleteHanlder function which changes the state of the component
* We can remove modal and backdrop from App.js
* Instead, we’ll import Modal and Backdrop in the Todo.js file
* Now, if the modal is meant to be open, we want to display the Modal element, and null otherwise



* The above code is equivalent to using the following, but Backdrop instead of Modal. Using the logical and operator, if both conditions are true, the second value will be executed. The component function will always be true, so it depends on if modalIsOpen is true.



Text

Description automatically generated

* Now, if we run our program, the Todo component will execute but since the state for modalIsOpen is false, the Modal and Backdrop app are not displayed. Once we click the button, it references the deleteHandler function which will set ModalIsOpen to true. This will re-execute the Todo component and since modalIsOpen is true, the Modal and Backdrop elements will be executed and displayed.

**Event Props**

* We want to close the modal when the user clicks outside the modal(backdrop), cancel, or confirm, but not the modal itself.
* Thus, we might try to create a closeModalHandler function that changes the state of the component, namely closing the Modal. This closeModalHandler function will be called by clicking on the Backdrop.
* HOWEVER THE BELOW CODE IS INCORRECT (as of now). It is incorrect as the Backdrop component is not a built-in component, so the onclick prop is not defined so clicking the backdrop never calls the closeModalHandler function. Thus, we need to add an onclick prop to the Backdrop component.
* Text

  Description automatically generated
* Below, we add props to the Backdrop component since we want to change the state of the Modal to false when the backdrop div is clicked. First, we add a props parameter object to the Backdrop component. We want to pass in a function that can accessed via props.onClick. Thus, in our div we can add an onClick that points towards the function that is passed in which is props.onClick. Note that the onClick in the div must be called onClick since it is a built-in prop. However, props.onClick could be called props.onCancel or whatever since it is a user-defined prop. So the closeModalHandler function is passed in as an argument for the Backdrop component, which the Backdrop component references when the Backdrop div is clicked. The Backdrop component references this argument function via props.onClick.
* Graphical user interface, text, application

  Description automatically generated
* The same idea is applied for the Modal component. In the Todo component, we want to change the state of the Modal to false when the cancel or confirm button is clicked. First, we add a props parameter object to the Modal Component. We want to pass in a function that is executed when the user clicks confirm and another function when the user clicks cancel. Thus, in the Todo component, we add the onConfirm argument and onCancel argument. These two arguments are both functions, in fact they are the same function as they both reference the closeModalHandler function. These argument functions are passed into the Modal component and accessible via props.onConfirm or props.onCancel. They are referenced when the user clicks the confirm or cancel buttons respectively. Note that it must be called onClick for the button prop since it is a built-in prop, while the onConfirm and onCancel props belong to the user-defined Modal component so we can name those props whatever.

Text

Description automatically generated

* Going back to the Todo component, it should look like the below
* Text

  Description automatically generated

**Routing**

* We are going to leave the todo app project and work on a completely new project so redo the installation process
* For the css, use this <https://github.com/academind/react-complete-guide-code/blob/zz-reactjs-summary/code/08-adding-routing/src/index.css>
* Make a components folder

A screenshot of a phone

Description automatically generated with medium confidence

* Routing is the url that is displayed
* Since we never fetch a new html page, we give the illusion of going to new pages by changing what is visible.
* We add a router which is a special tool that watches changes in the url and then changes what components to render and make visible on the screen depending on the url
* To add the routing functionality, we have to go to cmd in our react-project folder and run: npm install react-router-dom
* Make sure to run npm start after
* Create a new folder inside the src folder called pages. Just like the components folder, it is not mandatory but it is good style.
* Inside the pages folder, we will create new react components which is just like the components before, but will be loaded by the router. We will create the following files.

Text

Description automatically generated with low confidenceText

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

* Go to index.js and import BrowserRouter
* BrowserRouter is a component itself so we can use it as an html element and wrap it around the app component for example.
* We initialize the router package and make it aware of this app and changes to the url
* We have to define the urls we want to support and which components to be loaded based on these urls
* Go to app.js and import Route, and all the pages components
* Route is a component whose job is to define different paths and urls and the components that will be loaded based on these urls
* The part after the domain is the path. Ex ‘/hello’ is the path to ‘www.grant.com/hello’.
* We add the route component to the return and beside it we add path = ‘/insertPath’
* Inside the route component, we add the component that should be rendered

Text

Description automatically generated

* In the code above, if the path includes ‘/’, then the AllMeetupsPage will be displayed.
* However, if we go to <http://localhost:3000/new-meetup>, we get the following
* A screenshot of a computer screen

  Description automatically generated with medium confidence
* Both the AllMeetupPage and the AllMeetupPage components are rendered. React router matches all paths and checks if the current path starts/includes the paths. For example, <http://localhost:3000/new-meetup> has a path of ‘/new-meetup’. This path starts with ‘/’, hence why AllMeetupPage component is rendered. But the path also contains ‘/new-meetup’ which is why the AllMeetupPage component was rendered.
* Often times, this is not what we want, thus we also have to import the switch component
* We can use the switch component to wrap our route components and this tells react only one of these routes should be rendered.
* Text

  Description automatically generated
* However, when we go to <http://localhost:3000/new-meetup>, only the AllMeetupsPage component is rendered. This is because Switch stops looking at other routes once it finds a match. And by default, the Route component checks if the current path starts/includes the Route component path. The path of <http://localhost:3000/new-meetup> is /new-meetup. This path starts with ‘/’ which is one of the Route component paths and hence renders the component that corresponds wit the ‘/’ path which is AllMeetupsPage.
* To prevent this issue, we can add the exact prop after the Route component path. This exact prop tells the Route component that it should not check if the current path starts with the Route component path, but rather exactly equals the Route component path.

Text

Description automatically generated

* Now, when we run the program, and go to <http://localhost:3000/new-meetup>, the NewMeetupPage component is rendered as desired.
* Right now, we are changing the url manually. In the future, we could have a navigation bar that makes changing the url without having to do it manually possible.

**Adding Links and a Navigation Bar**

* Go to components, and add a new folder called layouts and create a MainNavigation.js file inside it
* Text

  Description automatically generated with medium confidence
* We could use an <a href= ‘’> tag to add links, but that sends a request to the server to get a new html page which is not what we want
* To avoid a new request, we can import the Link component
* The Link component is the same as <a>, but it attaches a click listener to the anchor tag so it doesn’t send a new request to the server. Instead, it’ll just parse the url you want to go to, change it in the url bar, and load the appropriate components.
* The Link component has a ‘to’ prop which is the path the link takes you to
* Text

  Description automatically generated
* Now we can go to the app.js file and import it and add it
* Text

  Description automatically generated

**CSS Modules**

* We probably want different css files for our different components. As well, it would be nice to have our css file to only affect a certain component, and not another one, even if there is a supposed ‘name-conflict’
* CSS modules allow us to scope styles to components, allowing us to attach css files to specific components
* Inside the layouts folder, beside MainNavigation.js, create a MainNavigation.module.css file.
* Add the following code to the css file <https://github.com/academind/react-complete-guide-code/blob/zz-reactjs-summary/code/10-styling-with-css-modules/src/components/layout/MainNavigation.module.css>
* In the MainNavigation.js file, import the MainNavigation.module.css file.
* When we import, we import an object, in our case called classes. All the css classes defined in the css file will be properties of the imported object.
* Behind the scene, the classnames will be name unique per component
* Now, we need to add the desired object classes to the elements

Text

Description automatically generated

**Outputting Lists**

* Go to AllMeetup.js
* Create a const array of dummy data found on <https://github.com/academind/react-complete-guide-code/blob/zz-reactjs-summary/code/11-outputting-lists-of-data/src/pages/AllMeetups.js>
* In react, arrays are automatically rendered by react. So {[<li>Item1</li>, <li>Item2</li>]} would become <li>Item1</li>

<li>Item2</li>

* Recall map takes an array, such as DUMMY\_DATA, goes through each element in the array, and creates a new array with changes we made to each element. We want to change our array from an array of object to li elements as shown below. The word meetup represents each element in the DUMMY\_DATA array which are the two big objects. Then, we can specify which property of the two inner objects we want via meetup.propertyName.
* We have to note that each child in a list should have a unique key prop which is a requirement by react, we don’t need to understand this. We just need to know we have to add a unique item for each li item.
* Text

  Description automatically generated
* We want to add styling to the list, but we want to make it component specific, so we’ll create new apps
* We’ll create another folder inside components called meetups. Inside the meetups folder, we’ll create a MeetupItem.js, MeetupList.js, MeetupItem.module.css, and a MeetupList.module.css file.

Text

Description automatically generated

* The code for the two css modules can be found at <https://github.com/academind/react-complete-guide-code/tree/zz-reactjs-summary/code/12-adding-more-react-components/src/components/meetups>
* Go to MeetupItem.js

Text

Description automatically generated

* This component returns a list item with some divs and other built-in html elements to make a meetup item. Note that the specific content will be passed in via the props object and can be accessed via props.title or whatever.
* Go to MeetupList.js

Text

Description automatically generated

* This component returns a list of all the MeetupItems. It takes in a prop object which is passed from AllMeetup.js. One of the prop attributes is meetups (we called it meetups ourselves) which returns the const array where each element is a meetup item that contains id, image, address and other info. Since props.meetups is an array, we can apply the map function on the array which goes through each element of the array. We let meetup represent an item in this array. meetup is the meetup item that contains id, image, address and other info which we can add access via meetup.property. Each of these meetup.property is an argument for the MeetupItem component used to display the MeetupItem component.
* Go to AllMeetups.js, import MeetupList and use this component to display all the Meetups.

**Props Children**

* Suppose we want to create a reusable component that acts as a wrapper around other components and give those components some styling
* In the components folder, create a new folder called ui. In there, we can store general components that don’t belong toa specific feature but instead are used in different places of the app
* Inside the ui folder, create a Card.js and Card.module.css file
* Fill the Card.module.css file with this code <https://github.com/academind/react-complete-guide-code/blob/zz-reactjs-summary/code/13-building-wrapper-components-props-children/src/components/ui/Card.module.css>
* Go to Card.js

Text

Description automatically generated

* Now, we go to MeetupItem and we can import the Card component and wrap it around the div
* Text

  Description automatically generated
* However, when we run the program, we can’t see any of the meetup information

Graphical user interface, website

Description automatically generated

* This is because the Card component simply renders a div and doesn’t know what to do with any wrapped content, so it just doesn’t render the wrapped content.
* To pass wrapped content to the Card component, we use pass in a special prop to the Card component. This prop is called props.children. Every component receives the props.children prop by default and props.children holds the content that is passed between the opening and closing of a component tag.
* Go to Card.js and add props.children

Text

Description automatically generated

* We can also add more left/right space by creating a Layout.js and Layout.module.css file inside the layout folder
* In Layout.module.css, add the following code <https://github.com/academind/react-complete-guide-code/blob/zz-reactjs-summary/code/13-building-wrapper-components-props-children/src/components/layout/Layout.module.css>
* Go to App.js, delete the line with the MainNavigation element. Then add the Layout component.

Text

Description automatically generated

* Now, the content in between the Layout component is passed to the Layout component and processed inside the function in Layout.js.

**Adding a Form**

* We want the user to be able to enter data about a meetup. And this data is stored in backend database
* Thus, we need to create a form to allow the user to enter this data
* We will create a new component which renders a form, we will create it in the meetups folder and call it NewMeetupForm.js alongside a css module called NewMeetupForm.module.css which we can get from <https://github.com/academind/react-complete-guide-code/blob/zz-reactjs-summary/code/14-adding-a-form/src/components/meetups/NewMeetupForm.module.css>
* In NewMeetupForm.js, import the Card component to give it some styling and also import the css module.
* Add styling from the css module into the divs.
* For the meetup title, we want the input to be of type text which should be required to have some in-browser validation and give it an id.
* For the label element of Meetup Title, we can add an htmlFor which is equal to "title" to point to the input element with id title.
* We can add other inputs as shown below. Note for the button, we use classe.actions to style it. As well, we don’t say type= "button" in the button as that would submit the form and generate a new HTML request.

Text

Description automatically generated

* Go to NewMeetup.js which is where we want to render the form and add the NewMeetupForm component alongside its import
* Text

  Description automatically generated

**Getting User Input and Handling Form Submission**

* To listen to the submission, we can add an onSubmit prop on the form which points to the submitHandler function (which we will create). On default, a submit event will be emitted if you have a button in a form and that button is clicked.
* The default behavior for the browser would be that it sends a request to the server. We don’t want that default, but rather, have our own logic.
* In the submitHandler function, we will automatically get an event object parameter.
* event.preventDefault(); gets rid of browser defaults.
* Now, we need to read the entered values.
* Since we are only interested in the user input once, when the user clicks the submit button, we can use refs.
* Ref stands for reference and react allows us to set references to DOM elements.
* To set up a ref, we need to import the useRef hook from react.
* Now, we can create a reference object which we will store in a constant. For example, we can create const titleInputRef = useRef();
* Another special prop that is built into react and supported on all elements out of the box is the ref prop. We add it to the DOM element we want to refence and point it to the reference object we earlier created.
* <input type='text' required id = 'title' ref = {titleInputRef}/>
* In the submitHandler function, we can get the entered title through const enteredTitle = titleInputRef.current.value;
* We could change DOM elements with ref, but we should use state for that. For reading, we can use refs.
* Repeat and make more refs for other inputs.
* Now, we can create a new object that has a title, image, address, and description key whose values come from the refs of the corresponding DOM elements.
* For now, we will just print out meetupData as the focus of reactjs is not to write backend code. We do not directly send code from react to a backend due to the lack of security.

Text

Description automatically generated

**Preparing the app for HTTP**

* With react, we often need a backend api to send our requests so the backend expects data in a certain format, usually JSON.
* Due to security, we need a backend api and a backend server which on the server connects to the database and stores data in a database.
* We’re not going to write our own api, we’re going to use firebase as a dummy backend. Firebase contains a database and an api to which we can send requests.
* Go to <https://firebase.google.com/>
* Click get started, create project, name your project, you can disable google analytics as we don’t need it, and create the project.
* Go to build tab, click realtime database, then click create database, click next for region, and click test mode, then click next.
* Now, we should have a url which we can use to requests to an api. And behind the scenes, those requests will be parsed and sent to a database.
* Go to NewMeetupForm.js, add props to the component function and replace console log meetupData with props.onAddMeetup(meetupData). This means a function is passed into the NewMeetupForm component and we can point to the function via props.onAddMeetup. This function should take meetupData as an argument.

Text

Description automatically generated

* Go to NewMeetup.js, and create an addMeetupHandler function which is going to be passed to the onAddMeetup component.

Text

Description automatically generated

**Sending a Post Request:**

* In the addMeetupHandler function, we want to send an HTTP request by using the fetch function which is a built-in js function
* The first string should be a url and we can add segments after the url to make folders

Ex: we change change <https://react-getting-started-5d46a-default-rtdb.firebaseio.com/>

To <https://react-getting-started-5d46a-default-rtdb.firebaseio.com/meetups>

We also need to add json at the end of the link

<https://react-getting-started-5d46a-default-rtdb.firebaseio.com/meetups.json>

* By default, fetch sends a get request, but we need to send data to the server so we want to make it a post request.
* We send the data, meetup, in JSON format and some other header stuff which doesn’t really matter

Text

Description automatically generated

* Now if we try to add a new meetup, and go to firebase, we see the stored data

Graphical user interface, text, application, email

Description automatically generated

**Navigating Programmatically**

* To give the user some feedback that adding the meetup worked, we probably want to redirect the user to another page
* To do so, we can import the useHistory hook from react-router-dom
* We can create a useHistory object and store it in a constant which exposes certain methods to us which we can use to manipulate that browser history and navigate away
* Fetch returns a promise which resolves as soon as the fetch to firebase is done so we then add the keyword, then, after the fetch to tell react what to do after the promise is resolved
* Inside the then, we add history.replace('/'); where '/' is a path.
* Now, when we create a new meeting, it redirects us to all meetups.

Text

Description automatically generated

**Fetching Data**

* We want to replace the dummy data with data from the firebase database
* We want to send a request to the database whenever the AllMeetup component is being rendered. Thus, before we render all the meetups, we want to fetch the data.
* Go to AllMeetup.js,
* Add fetch() and use the same firebase url we used to post our data. Except this time, we are getting data, and request method by default is get so we don’t need to change the method.
* Since fetch returns a promise, we can add a then method to handle the response of the firebase data.
* The fetch function returns a response object by default and we want to read the data we got by using response.json and return in.
* But json will also return a promise so we need to add ‘then’ to the end of the json. Since response.json was returned in the then for the fetch, in the then for the json, we have a data object. In the end, we want to extract an array of meetups from the data object and pass it as an argument for the meetups prop in MeetupList component.
* However, fetch returns a promise, js does not wait for the promise to complete before we render the return which includes the MeetupList component. We cannot use async await on the AllMeetupsPage function since that would make AllMeetupsPage not a component.
* Thus, we need to return some temporary JSX code such as a loading spinner and once we have a response, we update the JSX code.
* We can change what is visible on the screen with state, so import the hook useState from react
* We can now create a const that determines if we are currently loading and waiting for the response. const [isLoading, setIsLoading] = useState(true);
* We initially set isLoading to be true since we are waiting for firebase to respond
* Once we get the response, we use setIsLoading to set isLoading to false since we are no longer waiting for a response. We use setIsLoading in the then where data is a parameter object.
* We also have some loading JSX code that is shown if the isLoading is true.
* We also need to create another piece of state to manage the array of meetups. const [loadedMeetups, setLoadedMeetups] = useState([]);
* Now, in the then where data is the object parameter, we can use setLoadedMeetups to set the loadedMeetups to whatever the retrieved data from the database was.
* We now pass loadedMeetups to the MeetupList component as an argument for its prop and now, we can remove all the dummy data.
* DO NOT SAVE THE CODE YET
* However, we currently have an infinite loop as state tells react to re-execute the component function. The problem is when we change the state, the component reloads, and the fetch function runs again, and when firebase responds, the state is changed again, so the component reloads, and so on.
* To solve this infinite loop, we use useEffect

Text

Description automatically generated

**useEffect**

* To solve the infinite loop problem from the previous section, we can import the useEffect hook from react
* useEffect allows us to run some code under certain conditions.
* fetch() always runs when the component is evaluated, but we want to restrict it
* we can call useEffect which needs two arguments, a function and an array of dependencies
* For the function parameter, we put the code we want to run sometimes which is the fetch code
* If we did not specify the second argument which is the array of dependencies, then the useEffect function would execute every time the component is re-evaluated.
* With the second argument, react will check the values you added to the array and compare them to their equivalence when this useEffect function was executed.
* For example, DON’T DO THIS, if we were to add isLoading into the dependencies array, whenever the value of isLoading changes, the useEffect function would execute. If isLoading never changes, the useEffect function would not execute again.
* You shouldn’t thing of which values will trigger your useEffect function. Instead, in your dependencies array, you should add external values your useEffect function relies on.
* In our case, there are no external values. The fetch function is a built-in browser function and other than that, we don’t use any state or prop values inside the useEffect function. Note that setIsLoading and setLoadedMeetups are technically external dependencies but these state updating functions are an exception. This is because react guarantees that these state updating function never change.
* In the useEffect function, we want to add in the top line that setIsLoading(true); since if we ever do execute the useEffection function, that means we are making a request to Firebase so we are again loading and waiting for the response from Firebase.
* Another problem arises because setLoadedMeetups(data); is incorrect since when we fetch meetups from Firebase, we don’t get an array. Instead, we get an object where the cryptic auto-generated ids act as properties.
* Graphical user interface, text, application

  Description automatically generated
* So we get back an object with two properties in this case (and the cryptic shit is their id) and nested inside these properties, we get nested objects with the actual meetup data.
* Thus, when we pass the data to the MeetupList component, we expect an array since we apply map to the argument is only possible on an array.
* Thus, we need to transform the data before we pass it to MeetupList.
* We can create a meetups array
* Then we loop through each key in the data object retrieved from Firebase, which are the random id’s
* Inside the loop we create a meetup object which we will fill up with a key-value pair. The key is id and the value is key(which is the id of the cryptic shit).
* Then we use the spread operator to copy all the key value pairs of the nested object, data[key], into the meetup object.
* Then we add the meetup, which contains data about a specific meetup, to the meetups array
* Now, we can save the file and notice that the webpage loads when fetching data.
* Note, we can change data in the database just by going to Firebase­

Text

Description automatically generated

**React Context**

* We want to add the favorites feature where when we click the favorites button, the meetup is appears in the ‘my favorites’ tab. As soon as an item is favorited, we want to be able to unfavorite it. As well, we want to indicate the number of favorites beside the favorite tab
* To do this, we need to manage some state that will affect more than one component, since our list of favorites is some state that should cause the UI to update(as there is a number indicating the number of favorites), and the meetups displayed will also change
* Since we have a state that affects more than one component, we need to be able to manage that state globally and distribution that state to different components
* Simply using useState isn’t one component doesn’t do the trick since that only affects one component.
* We could manage our Favorites.js component’s state in App.js. This state in App.js could be passed as props into all of our components that need the state. For example in App.js, the state(the number of favorites) could be passed to the Layout component, which can be passed to the MainNavigation component which is where we want to display the number of favorites. As well in the App.js file, we could have another piece of state that contains the array of favorites which can be passed to the FavoritesPage component which is inside the App.js file. This would work, but it has hella downsides as it’s hard to maintain.
* There are ways to manage application wide state which is the built-in react state management solution called context.
* To make this context, create a new folder in the src folder called store (store is convention since we set up the state store in this folder)
* Inside store, create a favorites-context.js file which is where the context will be created
* import { createContext } from "react";
* use the createContext() to do as the function name implies, create a context
* context is a js object which contains a react component, which we can store in a const which should be named starting with a capital (since components start with capitals by convention)
* createContext also takes an argument which is the initial value of the context/component by state. It can be any value such as an object with key-value pairs.
* Graphical user interface, text, application, chat or text message

  Description automatically generated
* However, we need a way to change the values of a given key inside the object. Thus, we need to create a component inside the file
* We will create the FavoritesContextProvider component which is a regular component but it has the job of providing the context to all the components that are interested in listening to the values. As well, this FavoritesContextProvider component will be responsible for updating the values of FavoritesContext.
* We will return <FavoritesContext.Provider></FavoritesContext.Provider>
* Provider is a component that is built inside the context object that we named FavoritesContext. Thus, we access it via FavoritesContext.Provider
* This Provider component needs to be wrapped around all the components that are interested in interacting with the context. Thus, if we wrap the all the Provider component around all the components in App.js, so all components have access to the context.
* Recall props.children holds the content that is passed between the opening and closing of a component tag
* Thus, we can add props to the FavoritesContextProvider component and wrap the Provider component around {props.children}. This means we can now wrap the FavoritesContextProvider around any other component and those components will be wrapped by context automatically.
* In the FavoritesContextProvider, we still have to manage our context data. We can manage that with state. Thus, when we change the state of the FavoritesContextProvider component, the FavoritesContextProvider component will be re-evaluated, causing all the components that are wrapped by FavoritesContextProvider to be re-evaluated as well.
* In FavoritesContextProvider, we will create an object called context which is passed to the Provider component as a value as a prop.
* Now, whenever the FavoritesContext change, we want it to cause context to change and the updated context value will be passed into the Provider component as a prop. And passing this updated value will cause all the components that are listing to be updated.
* To change context, we can use useState so import it
* This state will manage the array of favorited meetups
* const [userFavorites, setUserFavorites] = useState([]);
* Now, the context object we created which should hold the latest values should have a favorites and totalFavorites key (since these were the keys defined in the FavoritesContext).
* We set the value of the key, favorites, to be userFavorites (which comes from state) and the value of the key, totalFavorites, to be userFavorites.length (which also comes from state)
* Now, we need a way to change the state to change the value of userFavorites

**Different Ways of Updating State**

* To change the state within the FavoritesContextProvider component, we can use functions
* We can add addFavoriteHandler(), removeFavoriteHandler(), and itemIsFavoriteHandler()
* For addFavoriteHandler(), we will have a parameter, favoriteMeetup, which is a meetup that we want to add to favorites.
* Inside addFavoriteHandler, we could use setUserFavorites(userFavorites.concat(favoriteMeetup)) which adds an element, favoriteMeetup, to an array, userFavorites(which is a piece of state, hence the setUserFavorites function to change it).
* Text

  Description automatically generated
* However, react does not process state updates instantly. Because of that, when we try to update our state and since our new state depends on our previous state, it is possible that the last state update isn’t updated fast enough.
* Thus, an alternative to updating a piece of state that refers to the previous piece of state is to use a function which returns the desired value instead of passing the desired value directly.
* This function will need a parameter, prevUserFavorites, which is the previous state of userFavorites. React will ensure the functions are executed in the correct order so our state will update properly.
* Text

  Description automatically generated
* For removeFavoriteHandler, it will take a parameter, meetupId, which is the Id of the meetup that should be removed from userFavorites which is state that contains all of the favorite meetups. Then, we will filter out a the meetup inside userFavorites that has the id of the parameter .
* Text

  Description automatically generated
* For itemIsFavoriteHandler, it will take a parameter, meetupId, which is the id of a given meetup. The function checks if that meetup which corresponds to that meetupId is inside the array of meetups, userFavorites(which is state).
* Text

  Description automatically generated
* Text

  Description automatically generated
* However, these three functions are never called. Thus, we need to figure out where to call them.
* The three functions should be called from inside other components and pages
* Thus, we don’t want to just manage values that are accessible by different components, but also make functions that will change those values accessible to different components
* Thus, we will expose pointers to those functions to different components
* Inside the context, not only should we add favorites and totalFavorites, we should add key value pairs where the values are pointers to functions that will change the values of favorites and totalFavorites.
* Note, we don’t execute the function, we just point at it.
* Text

  Description automatically generated
* Now, we need to wrap out contextProvider component around all the components that want to interact with this context. And in those different components, we need to pull out the values/functions that we are interested in
* To do so, we need to export the function FavoritesContextProvider (provides other components with context data) and we should also export the FavoritesContext component as a default
* These two things are exported since we need to interact with both exports from outside the file
* Text

  Description automatically generated
* For the FavoritesContextProvider function, the myFavorites page and the navigation bar will be interested along with the MeetupItem components(since this has a to favorites button which should trigger to addFavoriteHandler function)
* Go to index.js and wrap everything around with a FavoritesContextProvider component after importing it
* import {FavoritesContextProvider } from './store/favorites-context';
* We wrap FavoritesContextProvider in curly brackets since we don’t want the default export(the context object), but rather, we want the component function instead
* And we import named exports between curly braces which is standard js
* Text

  Description automatically generated
* Now, go to MeetupItem.js and the To Favorites button. When this button is clicked, we want to add it to our context and we also want to update the text on this button depending on if the item is already in the context or not. If it is already part of the context, this button should no longer add the meetup to the context, but rather, remove it
* Thus, we need some logic when the button is clicked so we will create a function that handles the logic along with an onClick prop for the button that points to the function
* In the function called toggleFavoriteStatusHandler, we need data from the context. To do so, we use another hook by react called {useContext}
* import { useContext } from 'react';
* {useContext} which allows us to establish a connection between this component and the context by calling the useContext() function which takes a parameter of a context object which we can store in a const. Thus, we need to pass in the context object so we need to import it(we already exported it as a default)
* import FavoritesContext from '../../store/favorites-context';
* const favoritesCtx = useContext(FavoritesContext);
* Now, we can determine if an item is favorite by adding the following line
* const itemIsFavorite = favoritesCtx.itemIsFavorite(props.id);
* props.id is the id of the meetup that we want to check if it is in favoritesArray or not
* Recall we passed in the id of a meetup as a prop to the MeetupItem component so we can just use props.id as shown above.
* Now we can use the context inside the toggleFavoriteStatusHandler() function. The function simply adds a meetup(by using all the id,title,description… that were passed in as a prop) if it the meetup is not already favorites and it removes a meetup if it already is favorited.
* The toggleFavoriteStatusHandler will execute a function that changes the state of userFavorites in favorites-context.js, updating the FavoriteContext component and all child components, allowing all child compoennts to have access to the latest state
* Text

  Description automatically generated
* Now, we want to change the button to display Remove from Favorites' or 'To Favorites' depending on whether itemIsFavorite is true or false respectively.
* Text

  Description automatically generated
* Now, if we run our app, we can click the ToFavorites button and it will change. The logic works, but the visuals isn’t updated yet so we gotta change that.
* Go to Favorites.js and here, we want to display all the meetups that are favorited
* To do so, we need the meetups that are favorited which we will get from useContext
* Thus, we will import useContext like what we did for MeetupItem.js along with the FavoritesContext object from favorites-context.js
* import { useContext } from 'react';
* import FavoritesContext from '../store/favorites-context';
* Just like before, we again create a new const, favoritesCtx, to store the context.
* We also want to display some fallback content if no meetups are favorited so it doesn’t look like the page didn’t load. Thus, we will create a variable called content and let it store JSX code which is valid (it’s no different from returning JSX code)
* Text

  Description automatically generated
* However, one thing to note is that the meetups that are favorited will disappear after the page is reloaded since our context is only stored in memory(not permanently) and not in a database like our meetups
* We also want to add a badge beside the My Favorites tab indicating the number of favorited meets.
* Again, we import the following
* import { useContext } from 'react';
* import FavoritesContext from '../../store/favorites-context';
* Then we create a const to store the context and use totalFavorites from the context to use the total number of favorited meets

Text

Description automatically generated

Deployment:

<https://www.youtube.com/watch?v=dn4mmfbletg>