JSX – javascript XML

There are tags such as <div></div> which will give us normal HTML

Other tags like <Field /> leads to a functional component called Field

We start with index.html. This file usually contains script that links to bundle.js, which contains all the js.

In index.js, we look for “root”.

To begin a react project, we do the following procedures:

* node -v (to check version)
* npx create-react-app <project name> (e.g. npx create-react-app jsx)

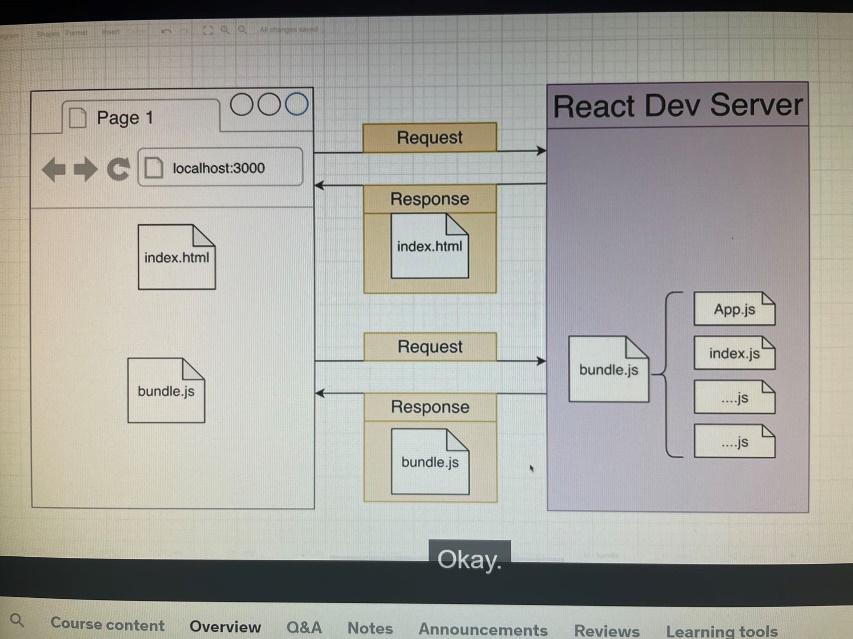
// below is procedure to start project

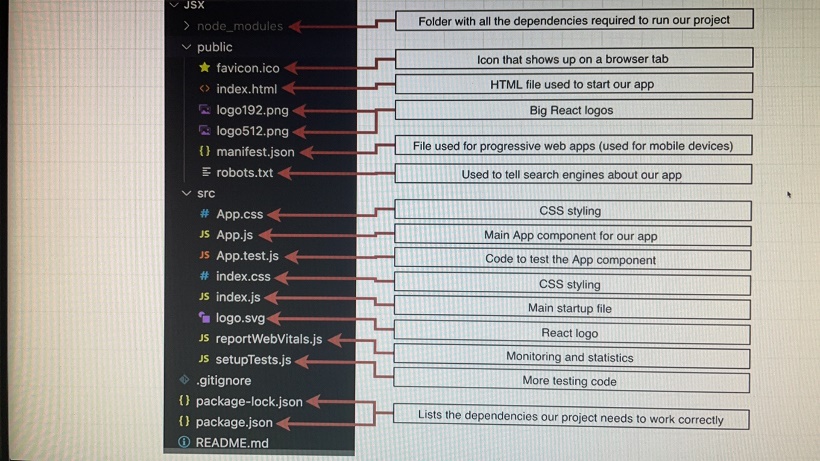
* cd <project name>
* npm start (this will automatically open a localhost react page)
* ctrl-C (to exit)

JSX is not valid javascript code. The browser does not know how to execute JSX. We need to transfile it before running the components. Transfile means take one form of code and turn into another type of code.

For example:

Babel (tool to turn JSX to js code) -> Webpack (tool to merge all files into bundle js) -> bundle,js





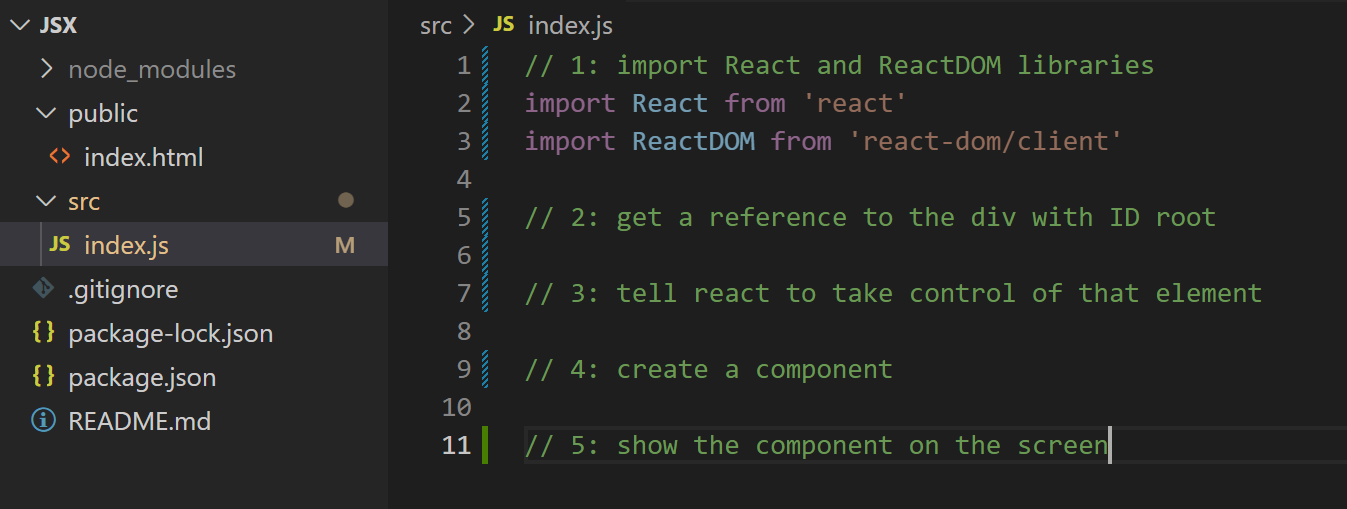
Not all of the files are necessary, most are just extra to make things look nice.

Required files:

* index.js (first file that gets executed when the app runs)
* index.html (skeleton for the react app)
* package.json (lists dependencies that our app needs)
* package-lock.json (same as above)
* node-modules (a folder with all of the required dependencies)

**Start of project:**

Create an index.js file



React is a library that defines what a component is and how multiple components work together.

ReactDOM knows how to get a component to show up in the browser.

(React is more of a base library, ReactDOM focuses on browsers)

Text

Description automatically generated

Create variables by calling “let”. Create constant variables by calling “const”.

Return the variable using { }. React cannot show an **object** as text content.

Text

Description automatically generated

We can also use { } for shorthand JS expressions.

Text

Description automatically generated

We can combine str text and variables in the return statement.

Graphical user interface, application

Description automatically generated

We can create input boxes. Either put in specified types, or use { }. The “type”, “max” are called properties. For numbers wrap in {}. For example, max={10}.

Text

Description automatically generated

Text

Description automatically generatedShape, arrow

Description automatically generated

Names and values of attributes in normal HTML are slightly different when writing in JSX.

5 rules to translate attributes in HTML to props in JSX:  
 1. In JSX, all props follow camelCase.

<input maxlength=“5” /> 🡺 <input maxLength={5} />

2. In JSX, all numbers should be provided with curly braces {}.

3. In JSX, “false” must use curly braces {}, though “true” can be ignored.

<input spellcheck=“true” /> 🡺 <input spellCheck />

<input spellcheck=“false” /> 🡺 <input spellCheck={false} />

4. In JSX, the “class” attribute is written as “className”

<div class=“divider” /> 🡺 <div className=”divider” />

5. In JSX, in-line styles are porvides as objects

style=“text-decoration: ‘none’; padding-top: ‘5px’;”

style={{textDecoration: ‘none’, paddingTop: ‘5px’}}

Usually we would like to place our App function outside of index.js.

* create App.js in src
* write function App, write “export default App;”
* Graphical user interface, text

  Description automatically generated
* Write “import App from “./App” ” in index.js file

There are 2 kinds of Export Statements: “default” and “named”.

A file can only have a single “default” export.

Default exports can be renamed in the importing file. E.g. “import MyApp from ‘./App’ ”

Example: “export default App”

Named exports are used to export multiple variables, and can have as many as we want.

Example: const message = “hi”

export { message }

import App, { message } from “./App”

Named export cannot be renamed.

“./” we are importing a file that we created, no “./” (like “react”) we are importing a package or module

**Reusable Components are useful**

create a reusable component (function)

component hierarchy is a diagram that helps us see the relationship between App component and the other components.

For example:

index.js App.js ProfileCard.js

shows App component on screen creates component with 3 copies of one instance of card

reminder: to create a react app, do “npx create-react-app xxxx”. “npm start” to start running app.

Common index.js code:

import React from "react";

import ReactDOM from "react-dom/client";

import App from "./App";

const el = document.getElementById("root");

const root = ReactDOM.createRoot(el);

root.render(<App />);

We use the props system to pass data from a parent to a child. This is one way flow, child cannot push data back to parent.

1. Add attributes to a JSX element
2. React collects the attributes and puts them in an object (props object) e.g. {color: ‘red’}
3. Props object shows up as the first argument to the child component function
   1. E.g. function Child(props) { return <div> {props.color} </div>; }

function App() {

    return (

        <div>

            <div>Personal Digital Assistants</div>

            <ProfileCard title="Alexa" handle="@1234alexa"/>

            <ProfileCard title="Cortana" handle="@cortana1234"/>

            <ProfileCard title="Siri" handle="@12siri34"/>

      </div>

    );

}

export default App;

function ProfileCard(prop) {

    return (

        <div>

            <div>Profile Card title is {prop.title} </div>

            <div>Profile Card handle is {prop.handle} </div>

        </div>

    );

}

export default ProfileCard;

We can use argument destructoring to make code look nice.

const { title, handle } = prop;

    return (

        <div>

            <div> Profile Card title is {title} </div>

            <div> Profile Card handle is {handle} </div>

        </div>

    );

We can simplify it even more. (When we don’t need the entire props object, but only a subset of the properties)

function ProfileCard({ title, handle }) {

    return (

        <div>

            <div> Profile Card title is {title} </div>

            <div> Profile Card handle is {handle} </div>

        </div>

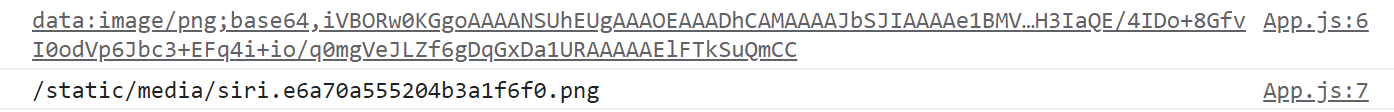
    );

}

Can use console.log(xxx) to debug

To apply images, we need to write “import xxx from ‘./sssxxxsxsxsxs.png’”, notice that .js files do not need to write .js at the end, where other files need to.

Depending on how big the images are, if larger than 9.7 kb, it will be considered as another file, else will be a “base64” string stored in bundle.js.



If images are online, do “<img src=’xxxx.xxxx.photosx.xx/xx/x/xxx/’>”

function ProfileCard({ title, handle, image }) {

    console.log(title, handle, "hey");

    return (

        <div>

            <img src={image} alt="pda logo" />

            <div> Profile Card title is {title} </div>

            <div> Profile Card handle is {handle} </div>

        </div>

    );

}

CSS is important for styling.

Bulma is a good CSS library (free styling).

Every block of CSS is going to have a selector (.xxx{}) and a collection of different CSS rules.

There are two main ways to appl CSS: use CSS library or write custom CSS.

To use Bulma library, run “npm install bulma”.

When importing from mode\_modules, we don’t need to use relative path.

Text

Description automatically generated

Main procedure of using Bulma: look at the doc of Bulma.io, copy the css code, make sure to have the same className. This will ensure to get the correct selector in “bulma.css”.

General Procedure:

1. Create index.js, App.js, necessary other components
2. Write the basic JSX code, obtain all necessary text, images, etc
3. Look into Bulma and find preferred styles
4. Apply code to the css styles

**More advanced projects (buttons, dynamic)**

React has two very important systems:

The Event System: such as detecting a user that is clicking a button

The State System: updating the content on the screen

Buttons are in the event system. We can use an “onClick” event name of a button, and assign it a function to run.

function App() {

    const handleClick = () => {

        console.log("Button was clicked!")

    };

    return (

        <div>

            <button onClick={handleClick}> Add animal </button>

        </div>

    );

}

Anytime we want to be notified about a click, type, drag, etc, we want to use the event system.

1. What event we want to watch for?
2. Create a function (usually called an event handler or callback function)
3. Name the function as “handle” + event, e.g. handleClick (convention)
4. Pass the function as a prop to a plain element, using correct event name, e.g. onClick, onMouseOver
5. Pass reference of the event handler function, do not call the function

General events doc: <https://reactjs.org/docs/events.html>

Very common events:

onClick, when user clicks something

onChange, when user types in a text input

variations of event handler calls:

    const handleClick = () => {

        console.log("Button was clicked!")

    };

Same as

    const handleClick = function() {

        console.log("Button was clicked!")

    };

Similarly we could do the below using arrow function

    return (

        <div>

            <button onClick={ () => console.log("Button was clicked!") }>

Add animal

</button>

        </div>

    );

Moving on to state systems: we need an update version (rerender) everytime something has changed.

State is data inside our components that changes as the user interacts with our app. When this data changes, React will update content on the screen automatically. One and only way to change content React shows.

Example of using state:

import { useState } from "react";

function App() {

    const [count, setCount] = useState(0);

    const handleClick = function() {setCount(count+1);};

    return (

        <div>

            <button onClick={handleClick}> Add animal </button>

            <div> Number of animals is: {count} </div>

        </div>

    );

}

We define a piece of state with the useState function.

const [count, setCount] = useState(0);

count is the piece of state, initially equal to 0, will change over time.

setCount is the Setter function, it is used to update the piece of state.

useState(0) the 0 here is the default value of the piece of state.

useState(x) returns a 2 element array. Const [a, b] is using array destructuring. X is any default value, can be number, can be string, etc.

Rerendering process:

When calling setCount, React will call App() again.

1. First time render:

const [count, setCount] = useState(0);

count is default value 0, and the 0 useState(0) will not be used in the future.

1. Second time render (when user clicks button)

Will trigger setCount(0+1), will rerender

Second time calling App(), count will now be 1

In general, when calling the setter function, React will rerun the entire component (will it rerun the entire app though?)

Example when passing empty array [] as default state:

function getRandomAnimal() {

    const animals = ["bird", "cat", "cow", "dog", "gator", "horse"];

    return animals[Math.floor(Math.random() \* animals.length)]

}

function App() {

    const [animals, setAnimals] = useState([]);

    const handleClick = function() {

        // this modifies a piece of state, we do not want to modify

        // animals.push(getRandomAnimal())

        setAnimals( [...animals, getRandomAnimal()] )

    };

Javascript map function is helpful with lists, mapping. It basically calls the inner function many times, and appends to a list, then finally outputs the list. Below, renderedAnimals is the list returned from animals.map.

const renderedAnimals = animals.map(

        (animal, index) => {return <AnimalShow type={animal} key={index}/>}

    );

To write custom css, create the necessary .css file, and import it.

Create a class name to the elements.

<div className="app">

            <button onClick={handleClick}> Add animal </button>

            <div>

                {renderedAnimals}

            </div>

        </div>

Then inside the css file, can start writing custom css:

.app {

    display: flex;

    flex-direction: column;

    align-items: center;

}

Can apply css to typical elements, like buttons.

button {

    background-color: lightgreen;

    border: 1px solid green;

    border-radius: 3px;

    font-size: 20px;

    padding: 10px;

    width: 30%;

}

.animal-list {

    display: flex;

    flex-direction: row;

    flex-wrap: wrap;

    justify-content: center;

}

**Search bar, using API, HTTP**

General version of search bar:

App

Search bar ImageList -> ImageShow components

Data fetching process:

Our app -> request -> Unsplash API (making an HTTP request)

Our app <- response (array or objects) <- Unsplash API (making an HTTP response)

React has no tools or functions for making HTTP requests, react only cares about showing content and handling user events.

HTTP request

* Request line (GET <https://api.unsplash.com/images/search> HTTP/1.1)
* Headers (Accept-Version: v1 Authorization: Client-ID ABC123)
* Body

=> SERVER =>

HTTP Response

* Status Line (HTTP/1.1 200 OK)
* Headers (Content-Length: 1000 Content-Type: application/json)
* Body ( [{id: ‘123’}] )

HTTP Methods:

1. GET get some information from the server
2. POST tell the server to create some new record
3. PUT completely update an existing record
4. PATCH partially update an existing record
5. DEL delete a record

Response status code (e.g. 200 OK) indicates whether the request was successful or not.

200 successfully got data 400 request is bad (incorrect syntax for example)

201 record was created 410 unauthorized, must provide authentication details

204 record was deleted 403 Forbidden, not allowed

301 URL you made request to has changed 404 Not found 500 Internal Server Error

HTTP request and response takes time, so in code, we need to consider this “time” issue. (async, await)

Do async function(), await axios.get(…)

Must also async / await in App() function too.

function App() {

    const handleSubmit = async (term) => {

        const result = await SearchImages(term);

        console.log(result);

    };

The Unsplash official API: <https://unsplash.com/developers>

Process is to sign up, get an account, create a new app on unsplash to get an access key.

Access key: z58kDpj6tto89ulkRyaKqxOGVlHrrlwh09hq5MsPr6U

Checking the unsplash api documentation, we can see that we need:

Our app => HTTP request GET <https://api.unsplash.com/search/photos?query=oceans> => UnspAPI

Location photo search

Authorization: Client-ID Access-key

HTTP response will be an array of objects

React itself has no functions for making HTTP requests, so we commonly use Axios javascript library, or Fetch function built-in to the browser. (npm install axios)

To use axios:

axios.get(url, {

headers: {xxx},

params: {xxx},

});

axios.get("https://api.unsplash.com/search/photos", {

    headers: {

        Authorization: "Client-ID z58kD..(access key)..6U"

    },

    params: {

        query: "cars"

    },

});

Import the api function in index.js

import axios from "axios";

async function SearchImages() {

    const response = await axios.get("https://api.unsplash.com/search/photos", {

        headers: {

            Authorization: "Client-ID z58kDpj6tto89ulkRyaKqxOGVlHrrlwh09hq5MsPr6U"

        },

        params: {

            query: "cars"

        },

    });

    console.log(response);

    return response;

};

export default SearchImages;

two things to improve: use a parameter for query term changes.

We probably only need the array data, so we return response.data.results

async function SearchImages(term) {

    const response = await axios.get("https://api.unsplash.com/search/photos", {

        headers: {

            Authorization: "Client-ID z58kDpj6tto89ulkRyaKqxOGVlHrrlwh09hq5MsPr6U"

        },

        params: {

            query: term

        },

    });

    // console.log(response);

    return response.data.results;

};

Sibling components cannot directly communicate (pass information), to share info between sibling components, we have to involve the parent.

App <searchImages(term)> <images>

SearchBar <term> ImageList <images>

We can do parent to child communication using props.

How do we do child to parent communication? Treat it like a normal event.

Quick example below:

import SearchBar from "./components/SearchBar";

function App() {

    const handleSubmit = (term) => {

        console.log("do a search with", term);

    }

    return (

        <div>

            <SearchBar onSubmit={handleSubmit} />

        </div>

    );

}

export default App;

handleSubmit is a function that lies in App (parent). We can use it as a prop to pass to SearchBar.

Currently App() consists of only one child component, which is the SearchBar.

A picture containing text

Description automatically generated

The SearchBar has a prop onSubmit.

function SearchBar( {onSubmit} ) {

    const handleClick = () => {

        onSubmit("cars")

    };

    return (

        <div>

            <input />

            <br></br>

            <button onClick={handleClick}> Click me </button>

        </div>

    );

}

export default SearchBar;

It has a input part and a button, and when onClick, it will trigger an onClick event that calls handleClick.

This is where the onSubmit comes in place; handleClick will call onSubmit, which is from handleSubmit from the parent App().

If we want to do something after the user hits enter, we can use <form>, which is an HTML thing. It has a built-in submit event when hit enter.

By default, the form submit event will try to request from

url.com?name=xxx&address=xxxx in this form, so to prevent this from happening, we can do event.preventDefault();

An event object is passed into an event handler automatically every time.

function SearchBar() {

    const handleFormSubmit = (event) => {

        event.preventDefault();

        console.log("hey hey tell parent")

    };

    return (

        <div>

            <form onSubmit={handleFormSubmit}>

                <input />

            </form>

        </div>

    );

}

Anytime we make a form control (text input, text area, checkboxes, radio buttons, etc), do the below.

How to handle text inputs:

1. Create a new piece of state (using useState())
2. Create an event handler to watch for the onChange event
3. When onChange event fires, get the value from input
4. Take that value from the input and update that to our state
5. Pass the state to the input as the value prop.

const [term, setTerm] = useState('');

    const handleFormSubmit = (event) => {

        event.preventDefault();

        onSubmitTerm(term);

    };

    const handleChange = (event) => {

        // console.log(event.target.value);

        setTerm(event.target.value);

    };

    return (

        <div>

            <form onSubmit={handleFormSubmit}>

                <input value={term} onChange={handleChange}/>

            </form>

        </div>

    );

After each rerender, our “term” will now be the updated value. We can now read it, update it, etc.

If we only want numbers:

setTerm(event.target.value.replace(/[a-z]/, ""));

Requirements for keys: the key prop is a special prop, it has its own definition.

1. Use whenever we have a list of elements (map function)
2. Add the key to the top-most JSX element in the list (critical, MUST be at the top most JSX element)
3. Must be a string or a number
4. Should be unique in the list
5. Should be consistent across rerenders

function ImageList( {images} ) {

    const renderedImages = images.map(

        (image) => {return (<ImageShow theImage={image} key={image.id} />);}

    );

**Booklist project, allowing creation, edit of books, editing Array**

Basic components:

BookList -> [BookShow] [BookShow] [BookShow] -- BookEdit

BookCreate (use forms)

Where do we want to create a state?

When a state is updated, it will rerender the component where it is defined, and all of its component children. So all we need to do is find the lowest common component, that is the component where we want to define the state.

When using <form onSubmit={handleSubmit}>, enter keyboard and submit button acts the same.

State updating:

If the reference to the new state is the same as to current state (meaning no new state creation, only “modifying” the current state), react assumes that no rerender is required.

Thus we need to create a new state (array) using []. […books, {id:99, title: “heyhey”}]

Do not directly mutate/change/modify arrays or objects when they are states. Create a new one.

Array:

Adding to the end

const [books, setBooks] = useState([]);

    const createBook = (term) => {

        const updatedBook = [...books, term];

        setBooks(updatedBook);

    };

Adding to the start

const [books, setBooks] = useState([]);

    const createBook = (term) => {

        const updatedBook = [term, ...books];

        setBooks(updatedBook);

    };

Inserting in a middle index

const [books, setBooks] = useState([]);

    const createBook = (term, index) => {

        const updatedBook = [

...books.slice(0, index),

term,

...books.slice(index)

];

        setBooks(updatedBook);

    };

Removing an element from an array of specific value

Will use filter(), a built in array method in javascript

Filter function takes in a function as an argument, and the function has 2 inputs, the element, and the index. (like looping over all instances, similar to how .map() works)

If filter function returns true, the value is added to the new array, else not added. Basically, filter keeps true.

const colors = ['red', 'green', 'blue'];

const newColors = colors.filter(

    (color, index) => {

        return color !== 'green'

    }

);

The filter function always returns a new array. We can also remove by index (or other properties)

function App() {

    const [colors, setColors] = useState(['red', 'green', 'blue']);

    const removeColorAtIndex = (indexToRemove) => {

      // TODO: Remove the element at 'indexToRemove'

      // Don't forget to update state by calling 'setColors'

      const updatedColors = colors.filter(

          (color, index) => {

              return index !== indexToRemove

          }

      );

      setColors(updatedColors);

    };

Modifying elements in array. Use mapping function to find correct element.

books.map(

    (book) => {

        if (book.id == 2) {

            return { ...book, title: newTitle };

        }

        return book;

    }

);

Note that the mapping function creates new instances of arrays, so we are not modifying the old array, but completely creating a new one. In javaScript, all keys must be unique. Hence {id: 2, title: “hey”, title: “newHey”} will overwrite the old title, leaving us with {id: 2, title: “newHey”}.

Why not directly change, but do a new object create? This is because we need a new reference to a state in order for react to rerender.

Adding/changing properties to an object:

1. Create a new object
2. Copy paste all properties from existing object using …
3. Add the updated property, javascript will overwrite the old version

const changeColor = (color) => {

    const updatedFruit = {

        ...fruit,

        color: color

    };

    setFruit(updatedFruit);

};

Removing properties from an object: use destructoring.

const removeColor = () => {

    const { color, ...rest } = fruit;

    // rest object will have all other properties of fruit

    // except the color property

    setFruit(rest);

}

How to generate random ids?

Use Math.random(), will get a random decimal between 0 and 1.

Math.round(Math.random() \* 9999) this is not perfectly safe, but good enough for now

Use state to swap components. E.g. showEdit is true then show BookEdit component, else just show the title.

function BookShow( {book, onDelete} ) {

    const [showEdit, setShowEdit] = useState(false);

    const handleClickDelete = () => {

        onDelete(book.id)

    };

    const handleClickEdit = () => {

        setShowEdit(!showEdit)

    };

    let content = <h3>{book.title}</h3>

    if (showEdit) {content = <BookEdit />}

    return (

        <div className="book-show">

            {content}

            <div className="actions">

                <button className="edit" onClick={handleClickEdit}>Edit</button>

                <button className="delete" onClick={handleClickDelete}>delete</button>

            </div>

        </div>

    );

}

We can use <https://picsum.photos> to get random images, this is mainly useful in practice or testing stage. (this page is basically a lorem lpsum for photos)

If we are getting back the same images, then disable cache in inspect mode. This will allow browser to not assume you will get back same image from same website.

Graphical user interface, application

Description automatically generated

To try unique seed, we can do this (documentation in <https://picsum.photos> random seed)

<img alt="books" src={`https://picsum.photos/seed/${book.id}/300/200`} />

**Persisted with outside API (JSON-Server API)**

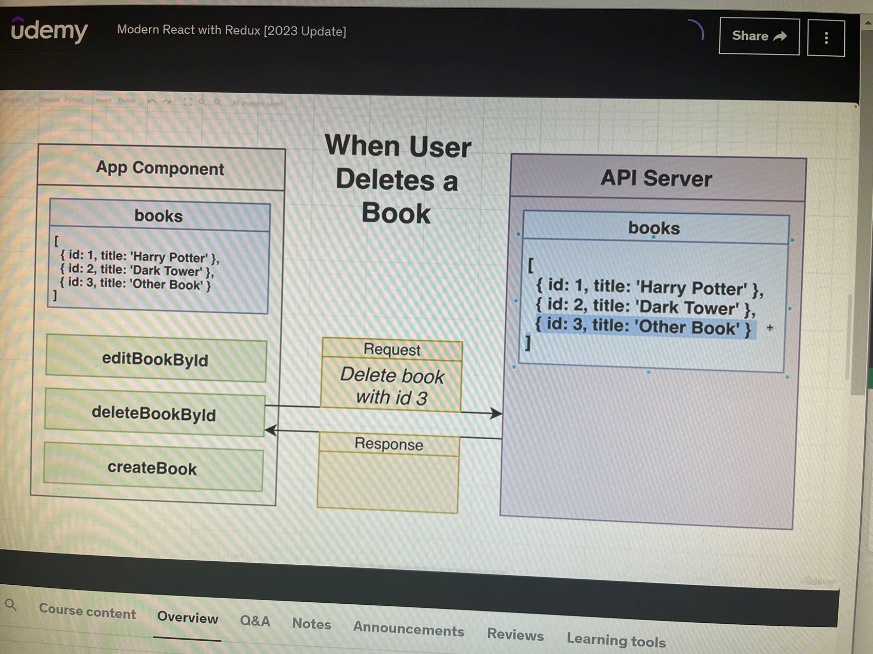
We currently have a local non-persisted list of books. Let us make a list of books persisted with outside API.

The basic problem is: if user refreshes the page, we will lose previous information. (0 persistence)

Solution:

React App -> request list of books -> API Server (JSON-Server)

<- response list of books <- ---- database

JSON-Server is an open source project for development and learning.

Modification on server list books, update to app component.

1. Create the JSON-Server API
2. When app starts up, we need to have an initial request to API to get current list of books
3. When user creates/edits/deletes a book, update the API, then update the app

JSON Server Setup

1. Inside the current directory, run “npm install json-server”.
2. Create “db.json” file outside src. This is where data will be store
3. Open package.json file, create a command to run JSON-Server

Text

Description automatically generated

1. Run the command. On another terminal, run “npm run server”

Text

Description automatically generated

From now on, need two commands to start this project working.

npm run start -> starts the react dev server

npm run server -> starts JSON-Server

Ports 2999

3000 ------ react dev server

3001 ------ JSON server

3002

"server": "json-server -p 3001 --watch db.json --host 127.0.0.1",

-p 3001 changes the port the server listens to to 3001

-watch db.json tells the server to store data in the db.json file

If we ever want to create a book, we send a POST request to <http://localhost:3001/books>, with object {“title”: “Harry Potter”}. This will be sent to JSON Server, which will add the object to db.json.

{

    "books": []

}

Will become

{

    "books": [

        {

            "id": 1,

            "title": "Harry Potter"

        }

    ]

}

It will automatically add the unique “id”.

JSON server will send back a response with the object with the id included. Goal of response is to say that record was saved and given id of 1.

Why <http://localhost:3001/books>? /books because our db.json has “books” as key.

To get all books, we call a GET request to <http://localhost:3001/books>. Will send back all books.

To edit books, we call a PUT request to <http://localhost:3001/books/1>, sending along the new object, and JSON server will notice the id 1 and update info with the new object. Will send back updated book.

To delete books, we call a DELETE request to <http://localhost:3001/books/1>, and server will notice the id 1 and delete object. Will send back deleted book.

Standalone API Client. It is a program used to make requests to an API server, specifically for development/testing. Vscode has a built in version. (REST Client extension).

Rest Client extension allows us to make requests off to some server either on our computer or hosted somewhere online. We need to create api.http file.

In the apt.http file, type, and then click send request

GET http://localhost:3001/books HTTP/1.1

Content-Type: application/json

Graphical user interface, text

Description automatically generated

Creating a new book, use POST

A screenshot of a computer

Description automatically generated with medium confidence

The ### is telling to divide the commands, this way it will know that they belong to different requests.

For the actual application, we will be using axios to do the requests.

const createBook = async (title) => {

        const response = await axios.post("http://localhost:3001/books", {

            title: title

        });

        // console.log(response);

        const updatedBook = [

            ...books,

            response.data

        ];

        setBooks(updatedBook);

    };

We would like to obtain the books every time we refresh the page. To do this, we write a function that initially “gets” the books from server, using axios.

    const fetchBooks = async () => {

        const response = await axios.get("http://localhost:3001/books");

        setBooks(response.data);

    };

Hard problem is, when should we call this function?

We use useEffect.

useEffect is a function that we import from react. It is used to run code when a component is initially rendered, or when it is rerended (if wanted). It takes in 2 arguments. First argument is a function, the function we want to initially run. Second argument is an array [] or nothing, used for rerenders.

    useEffect(() => {

        fetchBooks();

    }, []);

The function passed to useEffect will be always be called during initial rendering, and might be called (depends on second argument.)

[] – called after first render, never called again.

Nothing – called after first render, called after every rerender.

[counter1, counter2, …] – called after first render, called after rerender if counter variable has changed.

**Using the Context System**

We can use an alternative props, which is context. The goal is to reduce tediuous, we can share information through Context.

Context is not a replacement for props.

Context is not a replacement for redux.

A context object has two components: Provider and Consumer.

Provider is a component used to specify what data we want to share, and Consumer is used to get access to data.

import { createContext } from "react";

const BookContext = createContext();

<BookContext.Provider value={5}>

    <MyComponent />

</BookContext.Provider>

To share data, use the value props in context.Provider.

<MyComponent /> and its children can now access the value shared in context (5 in this example).

import { useContext } from "react";

import BookContext from "./book";

function SomeFunc() {

    const num = useContext(BookContext);

    return (

        <div>

            {num}

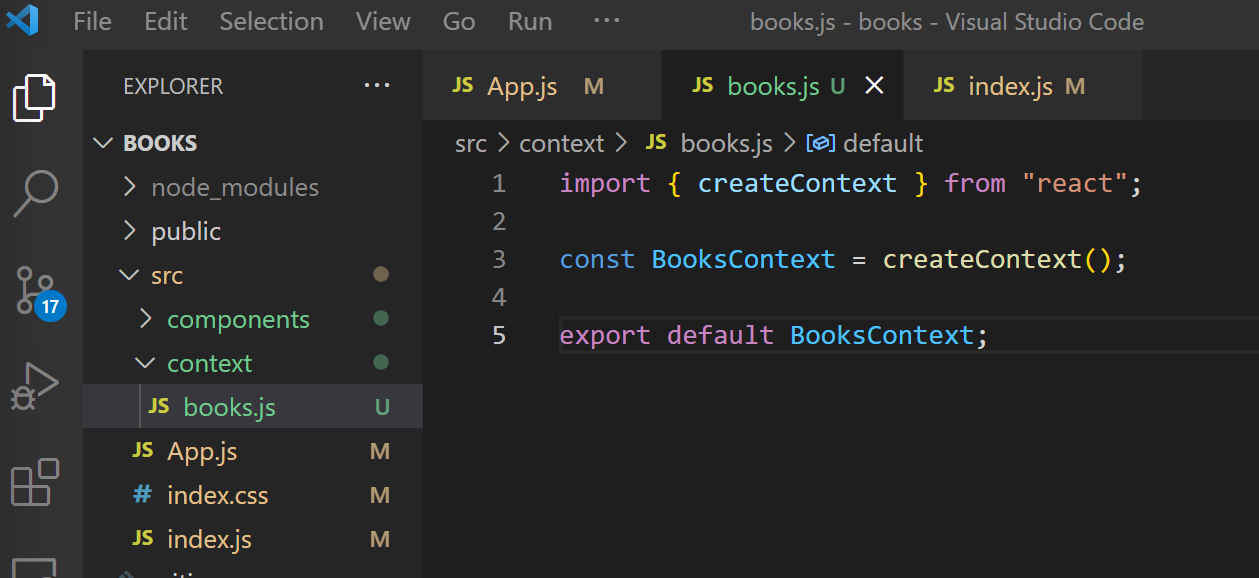
        </div>

    );

}

To get the values from context, call useContext(wantedContext).

We usually create a context file in a context folder in src.



We would then update our index.js file to make app() a component that receives context data.

Text

Description automatically generated

Currently, the value in shared context is static. We want dynamic.

(we need a second provider to wrap the first provider)

Text

Description automatically generated

Below is the code needed for this process.

For the BooksContext file, use createContext to create the context.

import { createContext, useState } from "react";

const BooksContext = createContext();

function Provider( {children} ) {

    const [count, setCount] = useState(5);

    const incrementCount = () => {

        setCount(count+1);

    };

    const valueToShare = {

        count,

        incrementCount,

    };

    return (

        <BooksContext.Provider value={valueToShare}>

            {children}

        </BooksContext.Provider>

    );

}

export { Provider };

export default BooksContext;

For index.js, apply the Provider function.

import { Provider } from "./context/books";

const el = document.getElementById("root");

const root = ReactDOM.createRoot(el);

root.render(

    <Provider>

        <App />

    </Provider>

);

For the other components where we want to use the BooksContext, we import and useContext.

import BookShow from "./BookShow";

import { useContext } from "react";

import BooksContext from "../context/books";

function BookList( {books, onDelete, onEdit} ) {

    const { count, incrementCount } = useContext(BooksContext);

    const renderedBooks = books.map(

        (book) => {

            return <BookShow key={book.id} book={book} onDelete={onDelete} onEdit={onEdit}/>

        }

    )

    return (

        <div className="book-list">

        {count}

        <button onClick={incrementCount}>click</button>

        {renderedBooks}

        </div>

    );

}

export default BookList;

Let us now look at other data we want to share across the app.

Two definitions:

1. Application state: data that is used by many different components
2. Component state: data that is used by very few components (local state)

Hooks

Functions given to us by react that add additional features to a component.

useState allows a component to use the state system.

useEffect allows a component to run code at specific points in time.

useContext allows a component to access values stored in context.

Custom Hooks

Functions we write to make reusable bits of logic, like similar to useState, useEffect, etc.

import BookShow from "./BookShow";

import { useContext } from "react";

import BooksContext from "../context/books";

function BookList() {

    const { books } = useContext(BooksContext);

change to below

import BookShow from "./BookShow";

import { useContext } from "react";

import BooksContext from "../context/books";

function useBooksContext() {

    return useContext(BooksContext);

}

function BookList() {

    const { books } = useBooksContext();

This is equivalent.

Next we put this custom hook into a new file in a new folder called “hooks”.

import { useContext } from "react";

import BooksContext from "../context/books";

function useBooksContext() {

    return useContext(BooksContext);

}

export default useBooksContext;

import BookShow from "./BookShow";

import useBooksContext from "../hooks/useBooksContext";

function BookList() {

    const { books } = useBooksContext();