# Q> **Define component lifecycle methods?**

A>

1. A component lifecycle method is a function that we can optionally define inside our class based component.
2. If we decide to define these methods, they will be called by react automatically at certain points during the component's lifecycle.

3. These methods are:

🡪 **componentDidMount** (content visible on the screen) --> **Good place to do initial Data loading**

🡪 **componentDidUpdate** (Sit and wait for updates) --> **Good place to do more data-loading when state/props change**

🡪 **componentWillUnmount** (Sit and wait until this component is no longer visible on the screen) --> **Good place to do cleanup (especially to non-react stuff)**

Technically

**CONSTRUCTOR** (**Good Place to do one-time setup, state initialize/data loading**)

and **render** (**Avoid doing anything besides returning JSX**) method is also a lifecycle method because it gets called at many times during the lifecycle of a component.

--> **Please make a note that** render method always gets called before componentDidUpdate method, it is because render method always returns some JSX which componentDidUpdate updates.

(you can check it yourself by console.log in render method)

**10 Aug 22**

**Good Practice**

Always prefer initial Data loading like API calls for geolocation APIs inside componentDidMount () instead of constructor (). It makes the code clearer and easy to understand by another developer. Use constructor method only to initialize state.

Q> **What are other lifecycle methods?**

A>

1. **shouldComponentUpdate**
2. **getDerivedStateFromProps**
3. **getSnapshotBeforeUpdate**

Q> **Can we initialize state outside constructor method as well in React?**

A>

Yes, we can initialize state object in the simplest way possible

state = { lat: null, long: null, errorMessage: "" };

Babel, the compiler that React uses, takes care of define constructor and super functions alongside state object.

constructor(props) {

    super(props);

    this.state= {lat: null, long: null, errorMessage: ""};

  }

It is same as this above implementation.

Q> **How do we pass state as a prop to child component?**

A>

1. Create an instance of child component <SeasonDisplay/>
2. Input State object’s property that we want to pass to this component as a prop. <SeasonDisplay lat={this.state.lat}
3. Notice we still using here **this.state** as a way to reference the property value.

<div>

          <SeasonDisplay lat={this.state.lat}/>

        </div>

1. So this way we can take state from one component and pass it down to a child component as a prop.
2. This property whenever it updates cause the Parent component to re-render as well as child component.

 (position) =>

        this.setState({

          lat: position.coords.latitude,

          long: position.coords.longitude,

        })

Q> **What is the default value of variables in JS?**

1. If we don’t assign a value to a variable in JavaScript, its value will be undefined.

You can check it by defining a variable like let name; and conole.log(name);

Q> **What are the best practices and rules of declaring variables in JS?**

* Cannot be a reserved keyword.
* Should be meaningful.
* Cannot start with a number
* Can’t contain a space or hyphen (-)
* Are Case sensitive
* We can define multiple variables in a single line , but it is preferred to do this in separate lines.

Q> **When do we get this error Uncaught TypeError: Assignment to constant variable?**

A>

If we define a constant variable,

let’s say **const interestRate = 0.3;** on one line

and in next line change its value by saying **interestRate=1.3**;

then we get this error because we can’t assign a new value to this variable.

Q>**How do we get current month from console?**

A>

Just write a simple code for this

**new Date (). getMonth ()**

It will return the current month (index -1). Because January starts from 0 and December ends in 11.

Q> **What’s the strategy to conditionally render items/components inside JSX?**

A>

1. Let’s take an example, that we want to display simple text either summer / winter on the basis of current month.

const text =season === "Summer" ? "Let's Hit the beach!" : "Burr..Its Chilly!";

1. Notice we put the entire statement inside another object. Which we can render inside a <h1>{text} </h1> tag like this.
2. Another use case is when we try to change a className of let’s say an icon tag <i> </i>

 const icon = season === "Summer" ? "sun" : "snowflake";

1. We put this inside an object as well which we can use this way when we return from JSX

<i className={`${icon} icon`}></i>

Q> **How to concatenate user defined string templates as className with inbuilt className in react?**

1. The className with which you want to concatenate your string should be put inside **` `** 🡨Back ticks

<div className={`d-flex justify-content-center ${showArrows}`} style={{ minHeight: '210px' }}></div>

1. The string you want to dynamically add with the current className should be put inside **{ }**
2. Add a **$** sign before curly braces.
3. Your string is now connected with current className (its ES-15 syntax).

**Comment what you see in the below with index**

const seasonConfig = {

  //4. Create an object called SeasonConfig

  Summer: {

    //5. Create array of objects inside it with Summer and Winter

    text: "Let's hit the Beach!", //6. Give these objects properties like text and iconName

    iconName: "sun",

  },

  Winter: {

    text: "Burr...Its Chilly!",

    iconName: "snowflake",

  },

};

const getSeason = (lat, month) => {

  // 1. Create a function called getSeason which uses lat and month as property

  if (month > 2 && month < 9) {

    //2. Check month prop if between month index 2 to 9

    return lat > 0? "Summer”: "Winter"; //3. Check lat and return "Summer" or "Winter" value

  } else {

    return lat > 0? "Winter”: "Summer";

  }

};

const SeasonDisplay = (props) => {

  //7. Our functional component SeasonDisplay

  const season = getSeason (props.lat, new Date (). getMonth ()); // 8. Store getSeason function's return value inside object called season

  const {text, iconName} = seasonConfig[season]; //9. season value is either "Summer" / "Winter”, so we put it inside seasonConfig object

  return (

    //10. to extract relative text and iconName property's values.

    <>

      <i className={`${iconName} icon`}></i>

      <h1>{text}</h1>

      <i className={`${iconName} icon`}></i>

    </>

  );

};

export default SeasonDisplay;

**Q>How to change style of a child tag from parent <div>?**

A>

1. Let’s take this for an example

<div className={`season-display ${season}`}>

    <i className={`${iconName} icon massive icon-left`}></i>

    <h1>{text}</h1>

    <i className={`${iconName} icon massive icon-right`}></i>

  </div>

1. <div className = {`season-display ${season}`}> is the **parent div** and <i className={`{iconName} icon massive icon-left`}></i> is the **child div** (<i> tag)
2. In CSS file we need to first locate the parent div and just add the tag value

. season-display. Winter i { 🡨 Add i tag without . (dot operator)

  color: blue;

}

. season-display. Summer i {

  color: red;

}

Q> **Show a loading screen in react**

A>

This is how we show a loading screen using semantic-ui

<div className="ui active dimmer">

        <div className="ui text loader">Loading...</div>

      </div>

Q> **If we forget to pass a value / miss a value to a prop inside a component, how to make sure it still gets a default value?**

A>

We can use **.defaultProps** as a component property and pass a default value to the prop inside it.

Take this component for example which returns a **“Spinner”** on page load with some Text message

const Spinner = (props) => {

  return (

    <div className="ui active dimmer">

      <div className="ui text loader">{props.message}</div>

    </div>

  );

};

If we don’t provide props.message , text will show Null.

So we decide to give this component’s prop “**message**” some default value.

Spinner.defaultProps = {

  message: "Loading...",

};

This way whenever we don’t provide message prop some value, it will always give “Loading…” as a Text rather than empty string.

But If we provide some value to the prop externally, it will always override the default.

**BAD PRACTICE**

Never Do Conditional Rendering inside the final JSX that you are returning. Take below example of a Bad code practice

 render () {

    if (this.state.lat || (this. state.long &&! this.state.errorMessage)) {

      return (

        <div>

          <SeasonDisplay lat={this.state.lat} />

        </div>

      );

    }

    if ((this.state.errorMessage && !this.state.lat) || this.state.long) {

      return <div>Error: {this.state.errorMessage}</div>;

    }

    return <Spinner />;

  }

}

This render function will return only one div or component based on the condition. What if we want to display a common background for all the returned items?

**GOOD PRACTICE**

Use **a helper function** instead, which handles the conditional rendering and call that inside final JSX render div (or from inside a container).

 renderContent () {

    if (this.state.lat || (this. state. long &&! this. state. errorMessage)) {

      return (

        <div>

          <SeasonDisplay lat={this.state.lat} />

        </div>

      );

    }

    if ((this. state. errorMessage &&! this.state.lat) || this. state. long) {

      return <div>Error : {this.state.errorMessage}</div>;

    }

    return <Spinner />;

  }

  render () {

    return <div className="border red"> {this. renderContent ()} </div>;

  }

}

The helper function in our case is renderContent () which we finally call from inside our final JSX div.

**17 Aug 2022:**

**Exercise:🡪**

* Refactor the UserForm to be a class-based component.  It should return the exact same JSX.

Remember that class-based components must:

1. Be a JavaScript class
2. Extend React. Component
3. Implement a render method that returns some JSX

const UserForm = () => {

return (

<form>

<label>Enter a username:</label>

<input />

</form>

);

}

**Solution**: After Refactoring to Class Based component:🡪

class UserForm extends React. Component {

render () {

return (

<form>

<label>Enter a username:</label>

<input />

</form>

);

}

}

Q> **Create a clock which display current time and updates every second**

A>  
Create a State object, where we will store our current time and which will be updated every second.

 state = { time: "" };

* 1. Inside **componentDidMount** lifecycle method, call a **setInterval method** (a method that executes a return statement after a particular time interval, Its 1000ms in our case)
  2. Now you can return setState method value from setInterval, which is going to re – run setState returned value every one seconds

 state = {time: ""};

componentDidMount () {

    setInterval (() => {

      this. setState ({time: new Date().toLocaleTimeString()});

    }, 1000);

}

* 1. Render this.state.time in final returned JSX
* render () {
* return <div className="time">The time is: {this.state.time}</div>;
* }

Q> **Show a basic text input field with label and some styling as quick as possible**

A>

I Used semantic UI to quickly get the styles needed for this purpose.

<div className="ui segment">

        <form className="ui form">

          <div className="field">

            <label>Image Search</label>

            <input type="text" />

          </div>

        </form>

      </div>

Q> **What is the difference between these two JSX expressions?**

A>

<input type="text" onChange={this.onInputChange} /> **//EXPRESSION 1**

&

<input type="text" onChange={this.onInputChange()} /> **//EXPRESSION 2**

* 1. In both statements, we can see that an input field is being rendered and we are using **onChange** handler method as a prop of <input/> tag for event handling.
  2. After that we are passing a callback function **onInputChange** into onChange prop, but I noticed that parentheses are missing in EXPRESSION #1
  3. By leaving off the parentheses, we are passing a reference of this function to the **<input/>** element
  4. This way this function is not called every time the component is rendered. **It gets called only when <input/> element decides to call it.**

**WE DO NOT PUT ON A SET OF PARENTHESES WHENEVER WE PASS A CALLBACK FUNCTION TO AN EVENT HANDLER**

**18 Aug 22**

Q>**Tell prop names for these operations?**

A>

1. User clicks on something 🡪 onClick
2. User changes text in an input field 🡪 onChange
3. User submits a form 🡪 onSubmit

We can use a callback function for these props as well, this way from inside the given prop. This syntax can be used **when we don’t want to declare a separate event handler function for invoking these events.**

<input

              type="text"

              onChange={(event) => console.log (event. target. Value)}

              onClick={(event) => console.log ("I was clicked")}

            />

**19 Aug 22**

Q>**What are controlled and uncontrolled components in the world of React?**

A>

1. **Controlled Components:** In React, controlled components are those in which form’s data is handled by the component’s state.
2. It takes current value through props and makes changes through callbacks like onClick, onChange etc.
3. A parent component manages its own state and passed the new values as props to the controlled component.

**Controlled Component example:**

const ControlledForm = () => {

  const [name, setName] = useState ("");

  function handleSubmit () {

    return alert (`Name: ${name}`);

  }

  return (

    <div>

      <h3>Controlled component</h3>

      <form onSubmit={handleSubmit}>

        <label>Name:</label>

        <input

          name="name"

          value={name}

          onChange={(e) => setName (e. target. value)}

        />

        <button type="submit">submit</button>

      </form>

    </div>

  );

};

B>

1. **Uncontrolled components:** Uncontrolled components are the components that are not controlled by the React state and are handled by DOM (Document Object Model)
2. So in order to access any value that has been entered we take the help of refs.
3. They are used for the cases where we need to add a file as an input which depends on the browser, so this is an example of uncontrolled input

**Un Controlled Component example:**

const UncontrolledForm = () => {

  const inputRef = useRef(null);

  console.log(inputRef);

  function handleSubmit () {

    return alert (`Name: ${inputRef. current. value} `);

  }

  return (

    <div>

      <h3>Uncontrolled Form</h3>

      <form onSubmit={handleSubmit}>

        <label>Name:</label>

        <input type="text" name="name" ref={inputRef} />

        <button type="submit">Submit</button>

      </form>

    </div>

  );

};

**DIFFERENCES🡪**

|  |  |
| --- | --- |
| **Controlled  Component** | **Uncontrolled Component** |
| The component is under control of the component’s state. | Components are under the control of DOM. |
| These components are predictable as are controlled by the state of the component. | Are Uncontrolled because during the life cycle methods the data may loss |
| Internal state is not maintained | Internal state is maintained |
| It accepts the current value as props | We access the values using refs |
| Does not maintain its internal state. | Maintains its internal state. |
| Controlled by the parent component. | Controlled by the DOM itself. |
| Have better control on the form data and values | Has very limited control over form values and data |

Q> **What is preventDefault () method in react?**

A>

1. The preventDefault method stops the default browser behavior of a selected element from happening by a user.
2. For example, prevent submit button from submitting the form whenever we click on it.
3. Or Prevent the link from opening after clicking on a Link URL.

handleFormSubmit(event) {

    event.preventDefault();

  }

  render() {

    return (

      <div className="ui segment">

        <form onSubmit={this.handleFormSubmit} className="ui form">

event.preventDefault() prevent the default behavior of browser refresh while submitting our form. Because we don't want browser to refresh after every submit action, we would rather run some of our own custom logic.

Q>**What is “*this*” in JavaScript?**

A>

**🡪“This” references the Object that is executing the current function🡨**

An Important rule of thumb:

🡪 If a function is part of an object, we call that function a method. So the first rule is “**In** **every method *this* references that object itself.”**

🡪Second rule is, If the function is just a regular function, which means it is not part of an object, then” ***this* reference to the global object which is the windows object in browsers and global object in node”**

**First Example**

Let’s start by creating a video object, in this object we are going to have a title property and a play method.

const video = {

  title: "a",

  play () {

    console.log(this);

  },

};

video.play();

When we call video.play(); we see our video object in the console.

*{title: 'a', play: ƒ}*

* 1. **play**: *ƒ play ()*
  2. **title**: "a"
  3. [[Prototype]]: Object

Since play () is a method in the video object, “***this***” references to object itself.

We can add another function in this object and will see the same result. Let’s try this as well.

video.stop = function () {

  console.log(this);

};

video.stop();

We create a new method stop() in video object and when we call video.stop(), we get reference to the same object in updated form.

*{title: 'a', play: ƒ, stop: ƒ}*

* 1. **play**: *ƒ play()*
  2. **stop**: *ƒ ()*
  3. **title**: "a"
  4. [[Prototype]]: Object

**Second Example**

Now let’s see what happens when we call a regular function and console.log(this) from there

function playVideo() {

  console.log(this);

}

playVideo();

It will return a global object, which is the window object in the browser

1. *Window {window: Window, self: Window, document: document, name: '', location: Location, …}*
   1. **alert**: *ƒ alert()*
   2. **atob**: *ƒ atob()*
   3. **blur**: *ƒ blur()*
   4. **btoa**: *ƒ btoa()*
   5. **caches**: CacheStorage {}
   6. **cancelAnimationFrame**: *ƒ cancelAnimationFrame()*
   7. **cancelIdleCallback**: *ƒ cancelIdleCallback()*
   8. **captureEvents**: *ƒ captureEvents()*

This window object contains contain a lot of property and methods available in the browser.

**Third Example**

What if we use a constructor function, what is ***this*** in constructor function?

We create a constructor function called Video() and pass a parameter title to it.

Then we use the constructor function to create a new object by using **new** operator.

function Video(title) {

  this.title = title;

  console.log(this);

}

const v = new Video("b");

Instead of window object, we get this Video object

1. *Video {title: 'b'}*
   1. **title**: "b"
   2. [[Prototype]]: Object

So whenever we use **new** operator, it returns a new empty object { }, and set in this constructor function to point to this empty object.

**Note**: if you call a function using **new** operator, which is the case for constructor function, ***this*** will reference a new empty object

Q>**What do you do, after getting TypeError: Cannot read property ‘state’ of undefined?**

A>

It’s a common error that we get when we are trying to get the state property from ***‘this’*** object, but if you try to console.log(this), you can see that it shows undefined in the console.

It’s because the value of ***‘this’*** is undefined from where you are invoking the method/function. So in order to give ***‘this’*** object the reference to an actual object, either use an arrow function syntax while declaring function/method or invoke a constructor function and. bind(this) with the function/method.

The goal of this activity here is to make ***‘this’*** not undefined.

Q> **What are *callback* functions, give some examples**

A>

In JavaScript, ***callback*** functions are also called higher-order function because,

In JavaScript functions are actually first-class objects, just like we can pass objects into functions as argument, you can also pass, functions into other functions as arguments as well, and execute them at some point.

**First Example**

let x = function () {

  console.log ("I am called from inside function x");

};

let y = function (callback) {

  console.log ("I am called from inside function y");

  callback ();

};

y(x);

Here, we have a function x and function y (which has a parameter called ***callback***, which is actually a function). This callback is called later in function y after console.log statement.

When we call function y with function x as an argument, we get



This example shows that, how a function x, gets executed inside of another function y as a callback argument. First function y gets executed and after that function x. So with same approach, we can execute something before or after the callback.

This example explains what a callback function is.

**Second Example**

This example will be used to demonstrate as to why do we need a callback function anyway.

let calc = function (num1, num2, calcType) {

  if (calcType === "add") {

    return num1 + num2;

  } else if (calcType === "multiply") {

    return num1 \* num2;

  }

};

console.log (calc (2, 3, "multiply"));

Initially, we have a function called calc which takes three parameters,

The first two num1 and num2 take numbers as argument while the third parameter calcType is about calculation type (add, multiply etc…)

Inside the function calc, it would be on the basis of calculation type, it would decide what to do with two numbers

What if instead of using string literal “add” and “multiply” as argument to calcType, this parameter was part of some library like jQuery and user is supposed to use this library and pass the number and calculation type to get the result.

This library might have all different kinds of mathematical operations or functions needed by the user and much more.

This is where the ***callback*** feature comes in picture.

Suppose a function name add is from a library

let add = function (a, b) {

  return a + b;

};

This function simply takes two arguments as numbers and return a value of their addition.

Now, instead of calcType, we can simply use a new parameter called ***callback*** and Inside calc function body we call this function and pass num1 and num2 as arguments.

let calc = function (num1, num2, callback) {

  return callback (num1, num2);

};

Now call the function calc with actual values.

console.log (calc(2, 3, add));

Now we understand ***callback*** function use here…

We can just simply use the function name in argument to execute a given operation. Another function that we created here doWhatever returns or simply print arguments passed inside it.

function doWhatever (a, b) {

  return console.log(`The numbers are ${a} and ${b}`);

}

let calc = function (num1, num2, callback) {

  return callback (num1, num2);

};

console.log (calc(2, 3, doWhatever));

**Output:** The numbers are 2 and 3

Summary of these examples, is that ***callback* is a great way to use functions from third party libraries.**

***Note:*** *You can pass callback function directly inside function argument without first defining it outside. It’s called* ***“Anonymous function”***

console.log (

  calc (2, 3, (a, b) => {

    console.log (a - b);

  })

);

**Output: -1**

***Note:*** *You can add some conditions before executing callback function. For example, when you need to make sure that User passes only a function as argument when expecting a callback function not any garbage value.*

let calc = function (num1, num2, callback) {

  if (typeof callback === "function") {

    return callback (num1, num2);

  } else {

    return console.log ("Error: Please pass a function only");

  }

};

*It passes and error message on console, in case argument is not a function.*

**Third Example**

Now, let’s look at a more practical example of a ***callback*** function.

var myArr = [

  {

    num: 5,

    str: "apple",

  },

  {

    num: 7,

    str: "cabbage",

  },

  {

    num: 1,

    str: "ban",

  },

];

This **myArr** is an array of string that we want to sort by strings first

Now JavaScript provides a built in sort method for arrays to do sorting

But it’s up to us on how we want this sorting to happen, for example in **ascending** or **descending** order of string.

So we will pass a ***callback*** function in the sort method to handle sorting type.

myArr.sort((val1, val2) => {

  if (val1.str > val2.str) {

    return -1;

  } else {

    return 1;

  }

});

According to this logic inside ***callback*** function, the str will be sorted in descending order

1. **0**: {num: 7, str: 'cabbage'}
2. **1**: {num: 1, str: 'ban'}
3. **2**: {num: 5, str: 'apple'}
4. **length**: 3

This ***callback*** is a very tiny function, but its deciding the sorting function output. We can not only sort strings but also numbers by just changing a tiny bit in the ***callback*** function

myArr.sort((val1, val2) => {

  if (val1.num > val2.num) {

    return -1;

  } else {

    return 1;

  }

});

1. **0**: {num: 7, str: 'cabbage'}
2. **1**: {num: 5, str: 'apple'}
3. **2**: {num: 1, str: 'ban'}

And Now, it’s in descending order of numbers.

Q> **How would you pass data, Let’s say from a child Class component to a Parent Class component?**

A>

To understand this process of conveying information backward from child to its parent **class component**, we need to look at an example and let’s try to do this in **4 easy steps:**

1. In the parent component **create a state object**, to store the data that we will get back from its child.
2. **Create a function with** ***callback*** (response from child) **as parameter** and **this. setState method** to update state object created in first step.
3. **Pass this function to child via props** <Child callback={callback}/>
4. Call this function in child component by using **this. props. function\_Name** and pass the data from child to this function as an argument.

*The parent component’s state will be updated when that callback function is executed in child*

**Steps from Code point of view: -**

Step 1:

state = { searchBarInputValue: "" };

Step 2:

handleSearchBarInputValue = (callback) => {

    this.setState({ searchBarInputValue: callback });

  };

Step 3:

<SearchBar searchBarCallback={this.handleSearchBarInputValue} />

Step 4:

handleFormSubmit = (event) => {

    event.preventDefault();

    return this.props.searchBarCallback(this.state.term);

  };

In the last step, when handleFormSubmit function will run after user submits his form, the searchBarCallback function is executed with updated value as the argument.

This response is sent back to handleSearchBarInputValue function, which is responsible for setState method to update parent component’s state object.

Q> **How network requests are generally handled in React applications?**

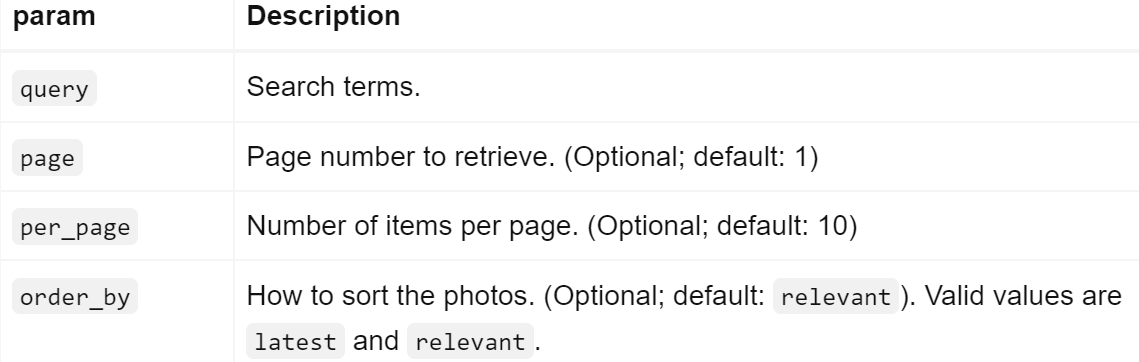
A>  
To make a network request to a third party API, It’s not going to be job of the React library itself.

1. To actually make a network request or a AJAX request to get some information, is going to be the job of some separate piece of code inside of our app.
2. The AJAX client is essentially the thing we are going to use to make a request over to the API.
3. React applications use **axios** {*a third party package*} or **fetch** {*function built into modern browsers*} for managing network requests and fetching some amount of data.
4. In most professional projects that I have been a part of, **axios** is being used because it handles network requests in a very predictable fashion.

Q> **How do you go through an API documentation, for example when you need to GET some photos by searching via a query on that API?**

A>  
*From the API documentation, these few things are needed to implement a basic API network request:*

1. **Type of Network Request** GET
2. In the Schema section, **Root/Base URL**: <https://api.unsplash.com/>
3. **API Endpoint** for selective data access /search/photos
4. If the API request requires **Authorization parameter in request body**, then find your ACCESS KEY in Authorization
5. If there is a **unique parameter** that we want to add in Network request body, say for search API we might need a parameter like query. An API might have several parameters that have different use cases (see image).



Q> **Based on the parameters found as per above question show a basic Axios get request?**

A>

1. To make a get request by Axios, we need to call axios.get() inside its parent function where we want to get the data from API.
2. This get () function, will take **two arguments**, the **address {“root URL + API endpoint”}** where we want to make request to and the second argument is going to be an {**object}** that will have a bunch of options (other objects) that will customize our network request.
3. Now we need to identify ourselves or who is trying to access the API, so we either need to pass our Access key via HTTP authorization header or via a Query parameter in our Network Request URL.
4. I go ahead with adding in a header of authorization, so in the second argument of axios.get(), which is an object. So one of the options that we can put inside this object, is a **headers** object.
5. This **headers** object is going to add some header on the request that we are making to this API.
6. This headers object will have a key-value pair

Authorization: “Client ID XXXXXXXXXXXXXXXXXX”

This will just add authorization header to the request, but we still have to actually make a request to get a particular result from the API.

1. In the API documentation, look under API **Parameters** where you will see a parameter’s name and its description which match your actual query.
2. So in order to add this parameter based on your query in axios, we will add one more option in the second argument of the get(), right with the headers object called params.
3. params specifies different query string parameters that we want to add into this request. In our case we want to add a key-value pair of {query: `some value`}
4. To actually see API in action, call axios parent function and inspect network – XHR to view the API, preview the data you received from the request.

  handleSearchBarInputValue = (callback) => {

    this. setState ({searchBarInputValue: callback});

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

      params: {query: callback},

      headers: {

        Authorization: "Client-ID HJqYXJQwC\_K7gvGwLbPAPmo4Nzk4mMN\_NjvdMrarxhk",

      },

    });

  };

We get below response from API network request:

1. results: [{id: "a4S6KUuLeoM", created\_at: "2018-08-05T12:50:37Z", updated\_at: "2022-08-24T18:04:17Z",…},…]
   1. 0: {id: "a4S6KUuLeoM", created\_at: "2018-08-05T12:50:37Z", updated\_at: "2022-08-24T18:04:17Z",…}
   2. 1: {id: "ZRns2R5azu0", created\_at: "2019-09-16T03:39:18Z", updated\_at: "2022-08-24T16:49:51Z",…}

Q> **Now use async-await syntax in the same example to retrieve API data**

A>

1. This way of using async-await syntax is a little bit easier for getting a little notification of when we get the response back from the API.
2. To use the async-await syntax inside handleSearchSubmit write **async** keyword in front of its name.
3. Now in front of **axios.get** Network request (*whatever taking some time to resolve*) write the keyword **await** and assign this to a variable called **response.**
4. You can console.log **response** value which contains API data.

async handleSearchBarInputValue(callback) {

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

      params: {query: callback},

      headers: {

        Authorization: "Client-ID HJqYXJQwC\_K7gvGwLbPAPmo4Nzk4mMN\_NjvdMrarxhk",

      },

    });

    console.log(response);

  }

Below is async – arrow function:

 handleSearchBarInputValue = async (callback) => {

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

      params: {query: callback},

      headers: {

        Authorization: "Client-ID HJqYXJQwC\_K7gvGwLbPAPmo4Nzk4mMN\_NjvdMrarxhk",

      },

    });

Q>**How can you contain all the code related to axios OR code related to Network request OR this network request’s configuration setting as a separate component and use it inside our main file?**

A>

1. Create a new folder inside our src directory and create a new file called unsplash.js (*I chose this name by my own convention because we are using the unsplash API*).
2. The goal of this component is to provide our API, headers and base\_url for now.
3. In unsplash.js file, I will first import **axios** from “axios”. One nicer thing about axios is that you can set up a *pre-configured instance* of the axios client
4. Axios client has *default properties set* for where it’s going to make a request to or headers and even params if you want as well.
5. One of the approach on how to do this, is to use axios. create method, which is going to create an *instance* of the axios client with a couple of default properties. *This customized instance is applicable for some specific URL with just these headers.*

export default axios. create ({

  baseURL: "https://api.unsplash.com",

  headers: {

    Authorization: "Client-ID HJqYXJQwC\_K7gvGwLbPAPmo4Nzk4mMN\_NjvdMrarxhk",

  },

});

1. Now rather than importing axios in the main file, we will import unsplash.js file now which is going to give us this *pre customized version* of the axios where we have *preprogrammed* our baseURL and the headers
2. Finally, we *replace axios with unsplash* in the main file.

handleSearchBarInputValue = async (callback) => {

    const response = await Unsplash.get("/search/photos", {

      params: {query: callback},

    });

Our code looks more understandable and less messy now.

Q> **Now you have successful API response, so what approach you will use to render the data in form of list?**

A>

1. I will create a separate ImageList component which will render out a list of images from response data.
2. After connecting this component to our main App component, I have to **convey/pass/communicate** the API response from parent to ImageList component. So I will use the prop system.
3. Now we pass a prop to ImageList called imageprop and assign it a value from parent’s state {this. state. images}.
4. To check if the props are successfully passed to ImageList, we can console.log (props. images) inside it.

<ImageList imageprop={this. state. listofimages} />

import React from "react";

const ImageList = (props) => {

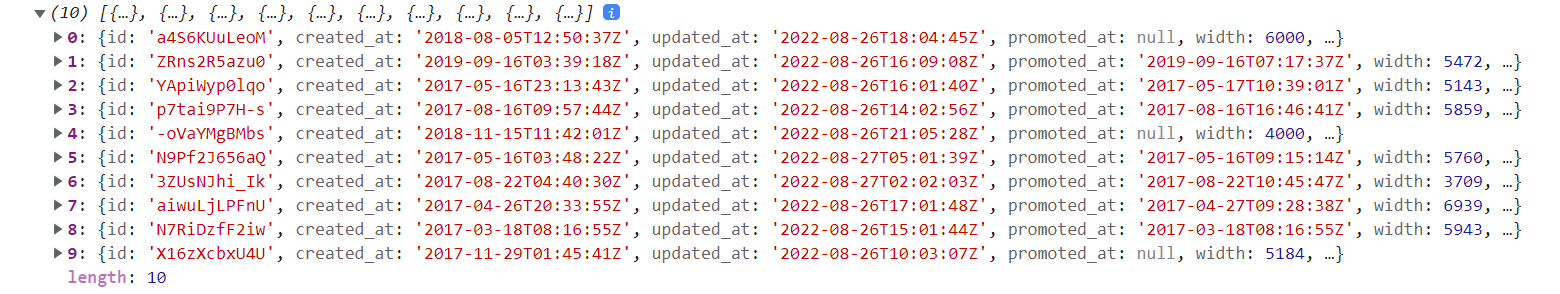
  console.log (props. imageprop);

  return <div>ImageList</div>

};

export default ImageList;

1. We get the list of images in the console.



1. Now we need to take this list of images and *turn it into actual list of elements* and show them to user.
2. First we will put a map function on props. images and use its returned object image to get the source URL.
3. Assign a constant to this map functions returned value so that we can put this inside our JSX.

const ImageList = (props) => {

  const images = props.imageprop.map((image) => {

    return (

      <li>

        <img alt="Not available" src= {image. urls. regular} />;

      </li>

    );

  });

  return <div>{images}</div>;

};

export default ImageList;

Images are now rendered in the form of list

Q> **Why do we need to add a key prop to the rendered list of items or Why we get this warning “Each child in array or iterator should have a unique “key” prop?**

A>

1. Suppose we rendered a list of 5 items on the DOM.
2. Now if I add a 6th item in the array, DOM will re render itself to display updated list with 6 items.
3. DOM will check each item’s unique key to make sure that this (6th item) is indeed a new unique item so it will re render itself accordingly.
4. So that’s the purpose of key prop, it is purely a performance consideration and it’s going to help react, render lists or updates to lists to be more precise and more performant.

const ImageList = (props) => {

  const images = props. imageprop. map (({id, description, urls}) => {

    return (

      <li key={id}>

        <img alt={description} src= {urls. regular} />;

      </li>

    );

  });

  return <div>{images}</div>;

};

Q> **map function exercises**

A>

1. Make an array of numbers that are double of the first array.

const numbers = [0, 1, 2, 3];

console.log(numbers.map((item) => item \* 2));

1. Take an array of numbers and make them into string.

console.log(numbers.map((item) => item.toString()));

1. Capitalize all letters of an array of strings.

const names = ["himanshu", "pandey", "happy"];

console.log(names.map((item) => item. toUpperCase()));

1. Display only object’s name property from an array of objects.

const names = [

  {

  name: "Angelina Jolie",

    age: 80,

  },

  {

    name: "Eric Jones",

    age: 2,

  },

{

    name: "Paris Hilton",

    age: 5,

  },

];

console.log(names.map((item) => item.name));

1. If age is less than 10, print they can enter matrix.

const names = [

  {

    name: "Angelina Jolie",

    age: 80,

  },

  {

    name: "Eric Jones",

    age: 2,

  },

  {

    name: "Paris Hilton",

    age: 5,

  },

];

console.log (

  names.map((item) =>

    item.age < 10

      ? `${item.name} can enter matrix`

      : `${item.name} can't enter matrix`

  )

);

1. Double the numbers of an array without using map function

const numbers = [0, 1, 2, 3];

const newNumbers = [];

for (let i = 0; i < numbers. length; i++) {

  newNumbers.push(numbers[i] \* 2);

}

console.log(newNumbers);

*As you can see, when we use map function we don’t have to initialize an empty array or use a for loop to go through each element of the array*

Date: 26 Aug 22

Q> **Explain the data flow while searching a term in Client Application to providing this term to API as query and as well as its reverse data flow when API provides response back to Client application?**

A>

1. Initially **Parent Component** is rendered with no list of images.
2. **handleSearchSubmit** function is called when user invoke submit action on the form.
3. Our axios **get** type network request inside handleSearchSubmit function gets triggered when its parent function is called.
4. We wait for a **response** from API.
5. We get the **response data** and **print it in console** to check.
6. In order for our parent component to re-render, we will call **setState** and set our response data on our component’s state.
7. This will cause our parent component to **re-render** and we can use this opportunity to **show a list of data**.

Q>**What are Promises in JavaScript?**

A>

1. First a little background on why Promises are needed in JS. We know that **JavaScript** is **single threaded**, which means everything including events, run on the same thread. If the thread is not free then code execution will be delayed, which can cause performance issues in our application.
2. So in order to handle **asynchronous tasks**, we need to do some **parallel programming** so that a unit of our code runs separately from the main application thread and **notifies it** in case either it has completed its task or not.
3. One way to handle asynchronous tasks in JavaScript is to use Promises. Promise as the name implies, is the function **“giving its word”** that a value will be returned at a later time. In other words, It’s a **proxy** for the response that we may/may not get in the future.
4. So instead of returning a fixed value, these asynchronous functions return a Promise **object**. This object has two main methods, **then ()** method which is executed when things go well and **catch ()** method, which is executed when the promise is rejected or in case of an error.

Q> **How to access Google’s YouTube API**?

A>Go to console.developers.google.com

1. Create a Project (A project is needed to view API’s and services).
2. Open the Created Project and click on Enable APIS and services.
3. It will open up a suite of APIS, you can search and select YouTube Data API and enable it.
4. Now Click on create credentials (To use API, we need credentials like API Access Key or OAUTH 2.0).
5. You can restrict key by applying Application restrictions (what webpages that can access this key, e.g. *localhost:3000*).

Q> **How can you solve; this issue cannot read properties of null (reading “snippet”).**

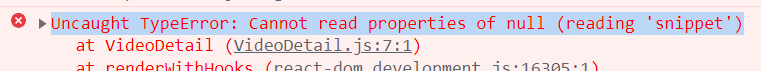
A>

1. This issue I faced constantly, when I was rendering a state object property in another component.

const VideoDetail = ({video}) => {

  return <div> {video. snippet. description} </div>;

};



1. The prop video is getting its value from a state object which is defined as null initially (selectedVideo: null)

state = { videos: [], selectedVideo: null };

1. selectedVideo gets updated only when user submits the form and trigger API response, until then it stays null.
2. So when I first load up my app, I get this error **cannot read properties of null reading (“snippet”).** Since API has not loaded up any results yet.
3. As a temporary workaround, we can do this before rendering video. snippet. description

const VideoDetail = ({video}) => {

    if (! video) {

      return <div>Loading...</div>;

    }

  return <div> {video. snippet. description} </div>;

};

Use an *If statement to return some <div> with text like Loading, in case our object is null*.

1. The Actual fix for this issue is using componentDidMount lifecycle method, it will be defined inside App component to do some default operation when the component first renders on the screen.

 componentDidMount () {

    this. handleSearchBarSubmit ("YouTube");

  }

1. It will provide this API handler function, with a default string, in our case the word ‘YouTube”. So that selectVideo gets updated as soon as the page is rendered

Q> What is </iframe>?

A>

1. It is like any other html tag, like <div> or <span> or <table> or <form>
2. It can be used to make a request to some outside website
3. It can make a request without any AJAX to a specific address.
4. Once it makes a request, website server will reply with some HTML and JavaScript to create a Video Player on the screen.

Q>How to use GIT properly in VS code?

A>

1. Open your project in VS code.
2. In the Left hand side below search option, we see source control.
3. Write “init” as a comment and click on commit.

Q> **What are hooks in React?**

A>

1. In simple terms hook system is all about giving functional components a lot of additional functionality.
2. Hook system has three main functions useState, useEffect and useRef.
3. useState allows us to use state in a functional component, very *similar to setState* function in class based components.
4. With useEffect we get access to something *very similar to life cycle methods* in a functional component.
5. useRef allows us to *create a ref or reference to a particular element* that is created by React.
6. In summary the hooks system is all about giving *tools to write reusable code, instead of using classic techniques like oop or inheritance.*
7. We can also create some *custom hooks*, which is basically a piece of code that does one very repetitive task and we want to make it as reusable as possible.
8. Inside these *custom hooks*, we may use our standard hooks like useState and useEffect.

Q> **How will you print index of each individual item of an Array?**

A>

1. When we apply map method on an array, the first argument is always the array item that we want to display as part of the list.
2. However, there is a second argument to map method which is an index and it can be used to display the *index or position of each individual item*.

 const renderedItems = items. map ((item, index) => {

    return (

      <React.Fragment key= {item. title}>

        <div

          onClick={() => console.log(`title ${index} clicked`)}

          className="title active"

        >

          <i className="dropdown icon"></i>

          {item. title}

        </div>

        <div className="content active">

          <p> {item. content} </p>

        </div>

      </React.Fragment>

    );

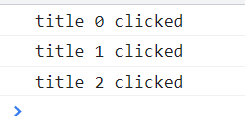
  });

  return <div className="ui styled accordion">{renderedItems}</div>;

};

export default Accordion;

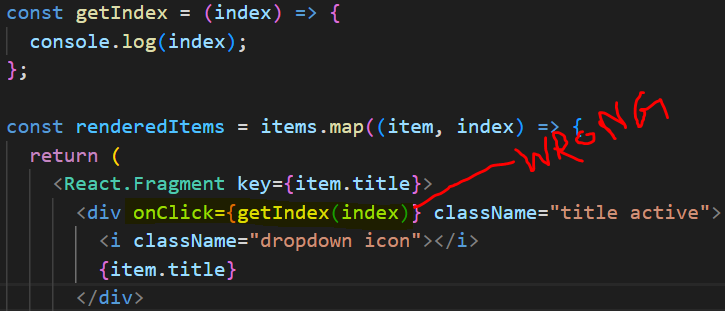
In the above example, when we click on a particular <div>, we get index of the div item that was clicked upon.



Q> **How do you call a helper function inside a component’s event handler**?

A>An event handler should use a helper function only when an event is triggered by user.

1. There for, there is a little *messy syntax* in react component to do this, so that an event handler is not called when the item is rendered initially
2. To do this a callback function syntax inside the event handler is used.
3. Here is an example for this to better understand.

Code Output

This getIndex(index) function is called when we refresh the page.

const renderedItems = items. map ((item, index) => {

    return (

      <React.Fragment key= {item. title}>

        <div onClick= {() => getIndex(index)} className="title active">

          <i className="dropdown icon"></i>

          {item. title}

        </div>

() =>getIndex(index) is the right syntax, only called when user invoke onClick event.

Q> **How to change className or apply styles in react dynamically?**

A>In below example we change the className from “title” to “title active”.

1. The reason behind it, is to show the content of this div *only when user clicks on it*.

 const renderedItems = items. map ((item, index) => {

    const active = activeIndex === index? "active":"";

    return (

      <React.Fragment key= {item. title}>

        <div onClick= {() => getIndex(index)} className= {`title ${active}`}>

          <i className="dropdown icon"></i>

          {item. title}

        </div>

        <div className= {`content ${active}`}>

          <p> {item. content} </p>

        </div>

      </React.Fragment>

    );

  });

 const active = activeIndex === index ? "active" : "";

In a constant, active we save the string “active” in case clicked item index and activeIndex (from useState) are same. This is true condition.

In case, it is not true we return empty string.

Lastly, we append the constant active in the className using ES2015 `*backtick*` syntax.

Q>**Explain useEffect hook in react and how to use it?**

A>This hooks allows functional components to use *something* *like* lifecycle methods.

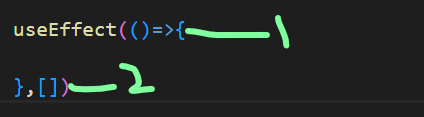
1. We *configure this hook to run some code automatically* in one of these three scenarios.

🡪 When the component is rendered **for the first time only**. (*when our functional component first appears on the screen*)

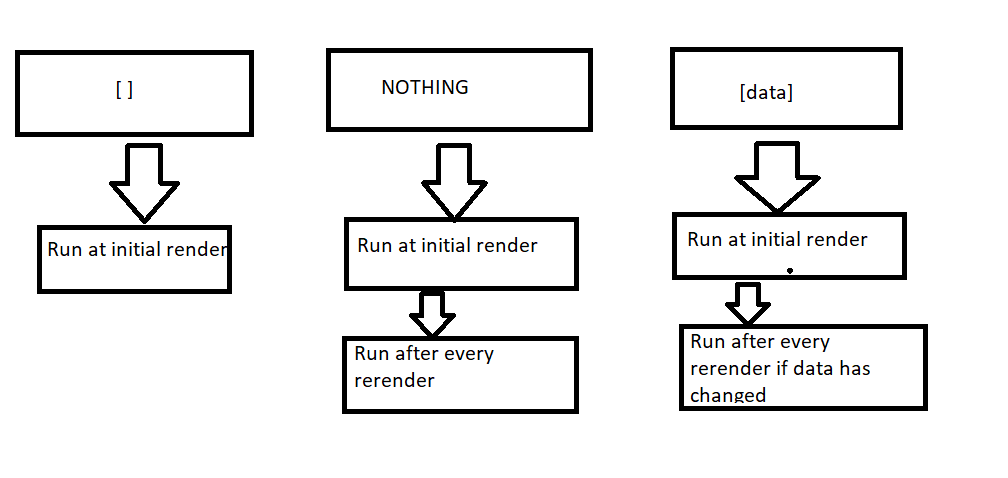
🡪When the component is **rendered for the first time and whenever it re-renders**. (*when our functional component first appears on the screen and then it re-renders*)

🡪When the component is **rendered for the first time** and **whenever it re-renders while some piece of data has changed**. (*when our functional component first appears on the screen and then some piece of state or data is changed and it re-renders*)

1. To understand this, first we need to understand syntax of useEffect.

🡨 useEffect takes two arguments

First argument is a function (an arrow function in my case) and second argument is an array. This second argument determines scenario for useEffect. *The three scenarios in which second argument can exist are no array or empty array or array with a piece of data inside it*.



The first argument of useEffect which is a function will run based on whether there is no array or empty array or array with a piece of data (*which changes in between re-renders*) as second argument.

1. It is a common practice to keep an empty array or array with some dependency inside it rather than leaving nothing as second argument.
2. This array can have multiple pieces of state inside it.

 const [searchTerm, setSearchTerm] = useState ("");

  useEffect (() => {

    const getSearchResult = async () => {

      await axios.get("https://en.wikipedia.org/w/api.php", {

        params: {

          action: "query",

          list: "search",

          origin: "\*",

          format: "json",

          srsearch: searchTerm,

        },

      });

    };

    getSearchResult ();

  }, [searchTerm]);

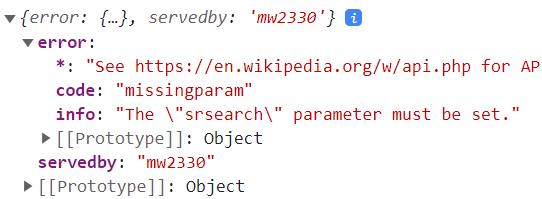
🡪In the above example, we define *async-await get request from Wikipedia API called getSearchResult* inside first argument of the useEffect which is expecting a function and then call *getSearchResult ()*.

🡪Now, when we first refresh the page, code inside useEffect is executed and we can see API response in our network tab.

🡪Notice we have a piece of state called *searchTerm* inside the dependency array of useEffect, so whenever there is a change in searchTerm function inside useEffect is executed.

**Note:**

🡪 Wikipedia API always require a value to its parameter *srsearch*, so we get an error when the useEffect code is executed on initial render.



To resolve this, we implement a condition inside useEffect to execute API request function *getSearchResult* only if there is a *searchTerm* entered by the user.

if (searchTerm) {

      getSearchResult ();

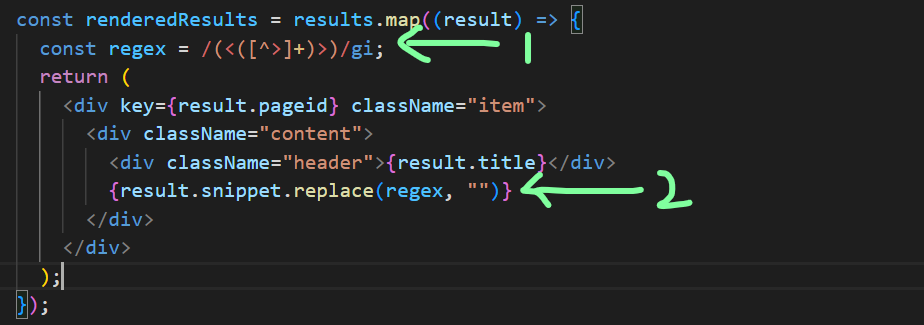
    }

Q>**How can we remove some HTML tags from an API response in React from rendering?**

A>  
**First Method:**

Use the /(<([^>]+)>)/ig regex with replace() method to replace HTML with empty string.

In this example we are getting HTML tags inside result. snippet which we don’t want.



so, we first define a constant called regex and then use it to replace all the HTML tags with empty string via replace method.

**Second Method:**

This is a kind of hidden method in the world of react due to its ability to make our app vulnerable to XSS (cross site scripting) attacks.

1. In the same example as for first method instead of displaying result. snippet inside plain curly braces, we are going to use <span> </span>

const renderedResults = results. map((result) => {

    return (

      <div key= {result. pageid} className="item">

        <div className="content">

          <div className="header"> {result. title} </div>

          <span dangerouslySetInnerHTML = {{\_\_html: result. snippet}}> </span>

        </div>

      </div>

    );

  });

1. We introduce a prop called dangerouslySetInnerHTML inside span tag which expects an object.
2. Inside this object, there we provide a key value pair of \_\_html: result. snippet (*YES, two underscores before html*).
3. Now It has given us the same result as first method.

Q> Why using <span/> with prop as ***dangerouslySetInnerHTML*** is not recommended by React?

A>

1. Anytime we take a *string from a third party API* such as Wikipedia API, we could be introducing a security hole into our application.
2. This security hole is specifically called an XSS or cross-site scripting attack.
3. This is where we accidently pick up or *render some HTML from an untrusted source* that can allow some hacker to execute some JavaScript inside of our application.
4. This untrusted source may be a server very much like our Wikipedia server which can send us fake set of search results, with some *malicious HTML and embedded inside is a JavaScript code*.
5. When this JavaScript code will run, it has the *capacity to replace our entire React app* with “HAHAHA, I CONTROL THIS APP NOW!!!”.
6. We give this opportunity to hackers as soon as we use dangerouslySetInnerHTML. So only if we are *confident enough on our HTML source*, we should use this.

Q> **After getting a list of results from the Wikipedia API, add a button to open a link based on a particular result from the list**

A>

1. Just add an anchor tag and provide a value to its href prop.
2. This href prop will use backtick syntax to append Wikipedia API url with pageid property of the selected item.

<a

        href= {`https://en.wikipedia.org?curid= ${result. pageid} `}

        className="ui button"

       >

This curid is from API documentation.

Q> **How to induce a wait for let’s say 500ms after a user stop typing in input field and then invoke the API request?**

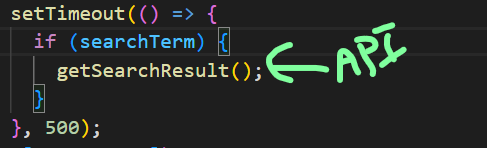
A>

1. We do not want to make an API request on every single onChange event (when user changes text input).
2. We need to allow user to type inside our input as much as they please and *not invoke API request on every key input* instead we wait for 500ms after last input from user is entered only then we will go ahead and invoke the actual API request.
3. The *mechanism of implementing* this is as follows

🡪 Whenever a user first types inside that input we are going to set up a timer using setTimeout method.

🡪 In that setTimeout function, we are going to say in 500ms, go and run the API request, with whatever user has entered in input field

*🡪 If user immediately presses another character, we are going to cancel the previous timer and set up a new timer to do a request in 500ms*. **This loop will go on till the last key press**, where user wait for 500ms without typing anything in.



*Step one of setting up our timer is done*. Now, our API gets called after 500ms of entering last input

🡪 *Step 2 is cancelling previous timer if there is an additional input change*. For this we have to first see how it actually works in our console.



🡪 First write a **setTimeout statement in the console** and observe that you get an identifier **“53”** in the next line.

🡪 This identifier gives us the opportunity to *cancel our timer by making use of another built in JavaScript function*

clearTimeout.

🡪 *Call this function clearTimeout and pass identifier* as an argument in our case, clearTimeout (53). This will cancel our setTimeout timer and function inside it will not be invoked.

Q> **How will you clear your setTimeout function for a API request inside useEffect hook**?

A>

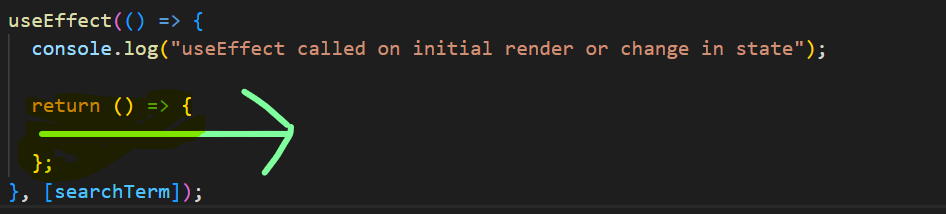
1. To understand how we can *cancel a timer inside useEffect*, we need to understand useEffect in more details.

  useEffect (() => {

    console.log ("useEffect called on initial render or change in state");

  }, [searchTerm]);

1. In this current implementation, useEffect will be called or more precisely the *first argument of useEffect which is an arrow function itself* will be called on initial render (first time our application refresh) or when we change our state viz. searchTerm.
2. Here is one thing to note at, when we provide this arrow function as first argument, there is only one thing we are allowed to return from this arrow function, Viz. **another arrow function itself!!**



1. The goal of this returned arrow function is basically to do some cleanup. Essentially, react call this function automatically and when we return this function react is going to keep a handle on it.
2. By keeping a handle, I mean, *react is going to keep a reference to this function and keep calling it automatically*.
3. Now understanding, at what point of time this function is called, is the whole challenge.

 useEffect (() => {

    console.log ("useEffect called on initial render or change in state");

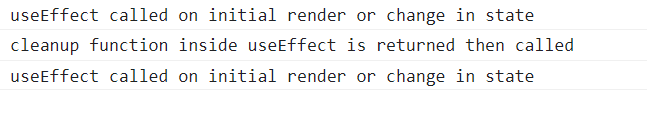
    return () => {

      console.log ("cleanup function inside useEffect is returned then called");

    };

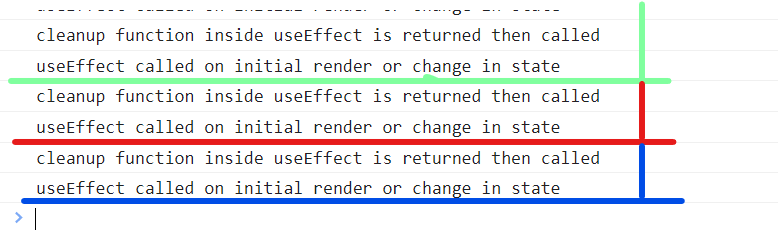
  }, [searchTerm]);

Take above code as an example, we get this output when our app refresh or at the time of initial render.

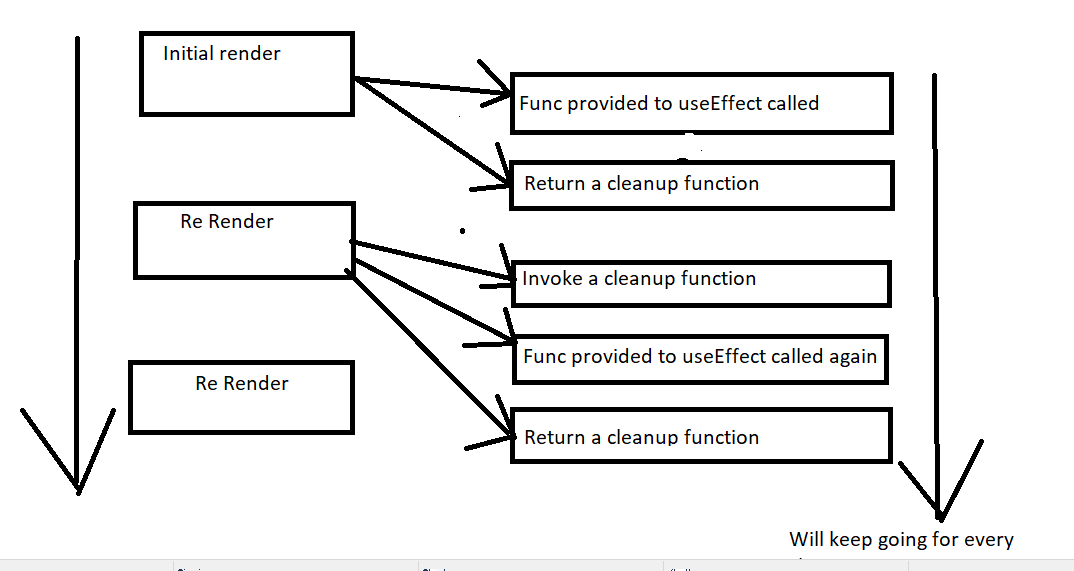


The overall arrow function (first argument) is run.

1. Now if we change our searchTerm, we see these two lines in our console, with every key press.



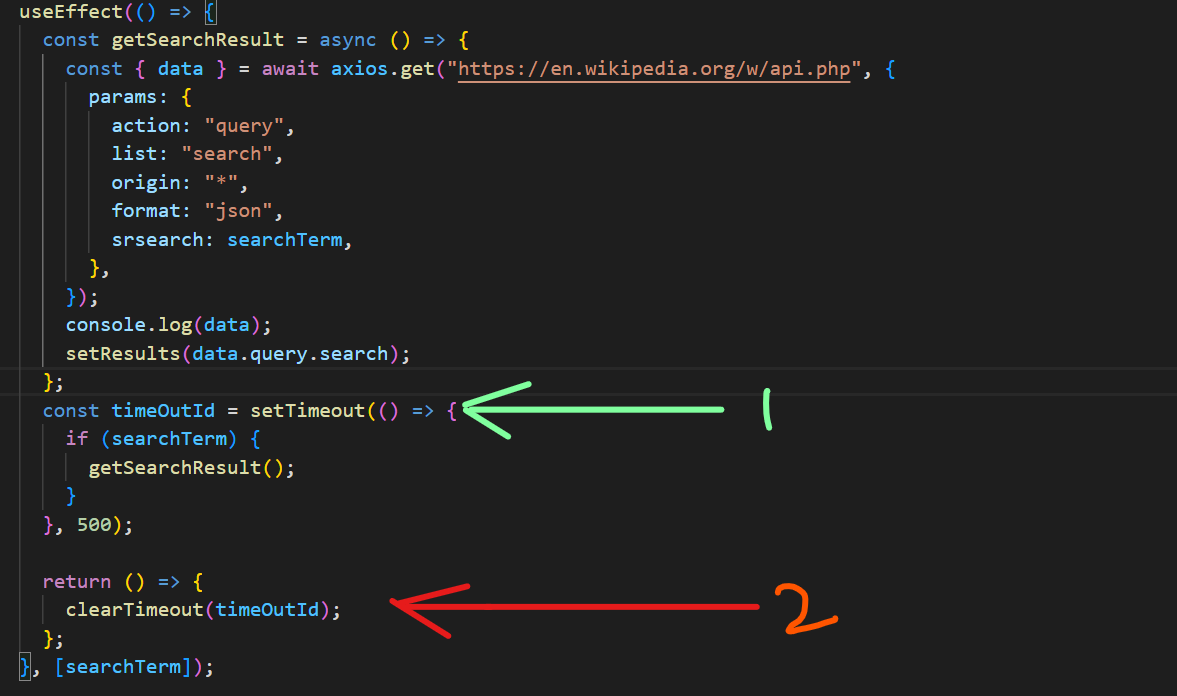
First cleanup function then first argument is called on each key press simultaneously.



*This is the purpose of cleanup function, to give us a little point on time to kind of do some cleanup or maintenance or just about anything we can imagine*.

This cleanup function is the key to how we cancel the previous timer.

In Code form, it will be implemented this way



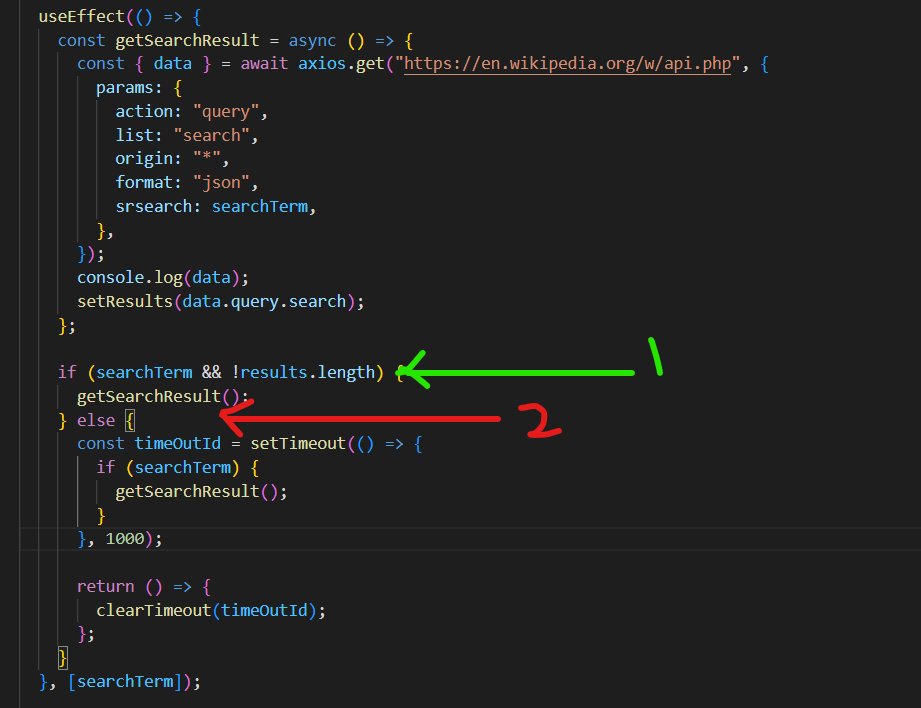
*First we get the reference to setTimeout identifier* which is timeOutId and then *pass this identifier as an argument to* clearTimeout method *inside useEffect’s cleanup function*.

Our timer in this case is 500ms, so if 500ms go by without useEffect being invoked, react is going to automatically execute the code inside first argument of useEffect, in our case its API search and setResults.

Q> This approach solves the problem of inducing a delay of 500ms on every key press or change in state, but it also causes delay in initial render as well, how will you solve that?

A>

1. The first time our application loads, we see our first search result only after waiting for 500ms.
2. What we need to implement is that, this delay should not be present when the first time our app loads. It should be only after when we start typing.
3. We need to add some checks to make sure this happens the right way.



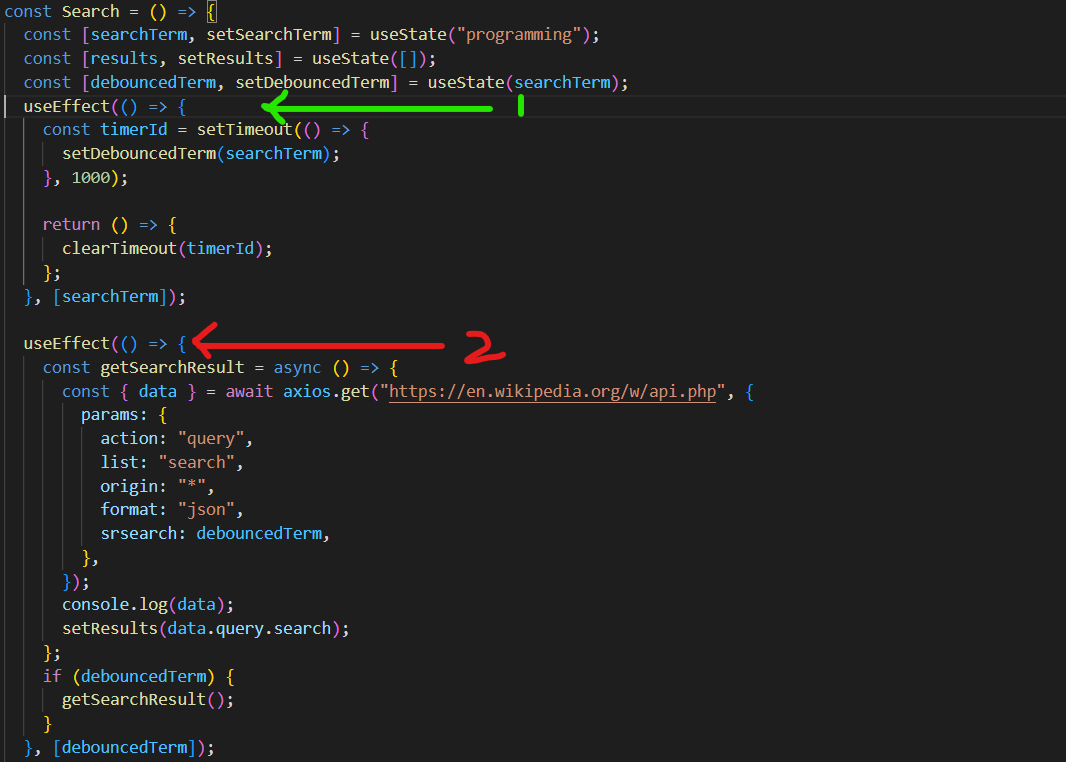
We add these **if-else** statements to handle this.

If 🡪 There is no searchTerm or no items in result array execute getsearchResult(*the API caller*) instantly.

Else🡪 Induce a setTimeout to *induce a delay* on getSearchResult and clearTimeout function to *cancel the timer*.

Q> **Give an example of using two useEffect hooks parallel or simultaneous to each other**

A>



The two useEffect hooks have two separate dependencies searchTerm and debouncedTerm.

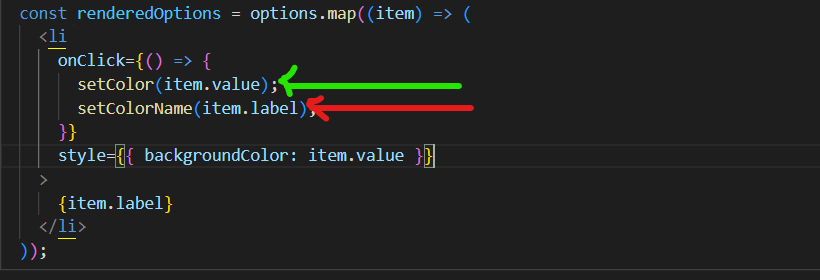
In first useEffect hook debouncedTerm is *updated after a delay induced by setTimeout and clearTimeout after getting searchTerm updated*.

*The second useEffect hook initiates API search only when debouncedTerm is updated*.

The combination of these two useEffect hooks makes sure these **3 advantages** as opposed to using single useEffect.

* At initial render there is no delay. It will be **instantaneous**.
* debouncedTerm is updated only after a **delay**.
* Same debouncedTerm in any case will **not initiate** **new API search**.

Note: Invoke two functions with a single onClick event **Syntax**

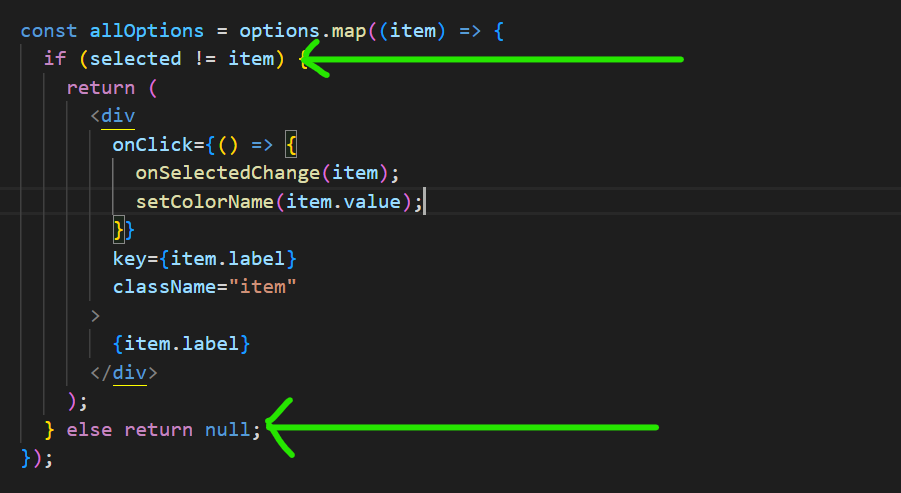


Q> How to show just those options from a dropdown which are not selected, in other words how to remove the option from dropdown after selecting it?

A>

Check in Dropdown.js

You can use a simple If-else statement to return JSX based on the condition.



If selected prop is not equal to dropdown item, then we return the JSX else we return null.

*Null in react means don’t render anything*.

Q> **How to close the dropdown menu when we click outside the component’s scope / rest of the application body / how to track click outside of scope**?

A>

*We can easily set up event handlers like onClick or onChange on the elements that are returned as JSX from Dropdown component using prop system*.

For example, when we click on that element we will run a function specified to that onClick event.

In other words, our Dropdown can only watch for clicks that happens on its elements.

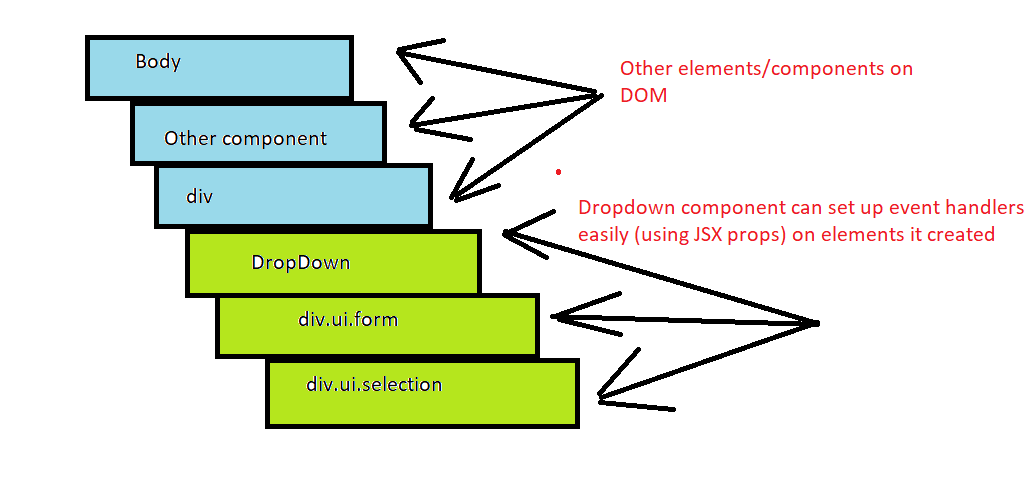
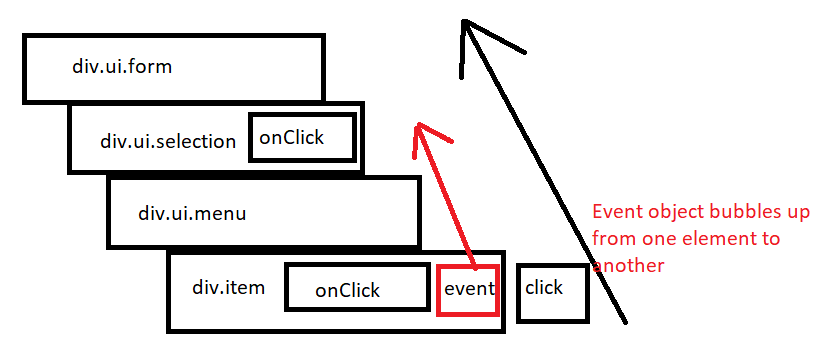


Fig. **HTML STRUCTURE**

The challenge is to listen to events that happen by *clicking* anywhere else on the screen. ***The entire event handling feature in “react” only allows a component to listen to clicks on elements that are created by that component***.

Before implementing our code, lets first discuss a topic around DOM called ***event bubbling***.

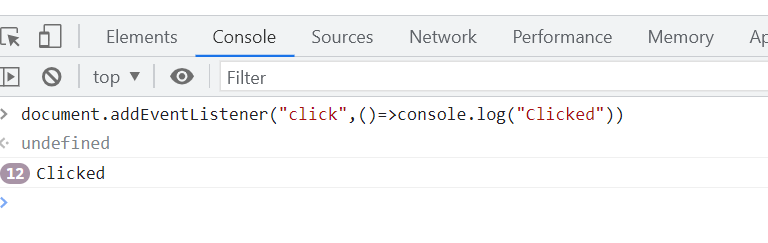


* Let’s imagine a user clicking on div. item, then the browser itself will create an event object.
* This event object describes some information about the click like mouse position, what element user clicked on etc. The *browser then hands that event object off to react*.
* React does some processing on that event and then *provide that event object to our onClick event handler as an argument*.
* The event does not stop just there, it then travels to next parent element, if it has some event handler it will be automatically invoked. This event object will keep moving on to all these successive elements and in *every step browser checks whether current element has an event handler or not*.
* Thus the name **event bubbling** as the event kind of bubbling or rising up our DOM structure.

With all these things in mind, we can create a workaround

The Dropdown can set up a *manual event listener* (without React) on the body element, then a click on any element will bubble up to the body

By Manual event listener, I mean this…



This is how we will detect clicks anywhere on the screen.

Now next step is configuring our Dropdown component to set up this event listener.

But, first note this…

Note: We see two type of event listeners in our app one is **document. body. addEventListener** (*event listeners wired up manually*) and **onClick** (*event listeners wired up by React*). **All the manual ones will get called first and only then our React event listeners will get called** (*from most child to most parent one-event bubbling*).

Now back to our code, we are going to trigger our manual event listener only when user clicks outside the dropdown, if we click inside the dropdown it should not trigger.

In simple words, need to figure out which element was clicked and whether or not that element was inside of our dropdown

Step 1: we will use (**event. target)** to figure out what element was clicked on.

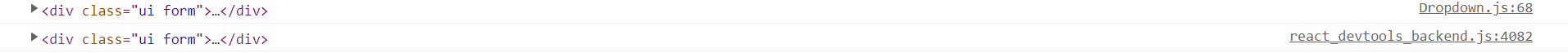
Step 2: Next, we need to figure out whether (**event. target)** is inside dropdown component or not. So we will use a hook called useRef. It will help us get a reference to a direct DOM element which we will eventually *use to figure if the element that we clicked on is inside that element or not*.

const ref = useRef ();

<div ref={ref} className="ui form">

We get reference to above element.

You can see console.log **(ref. current)** to see its current property.



Since we have element reference now, we can easily add a condition whether (**event. target)** is inside **(ref. current)** or not.

useEffect (() => {

    document. body. addEventListener (

      "click",

      (event) => {

        if (ref. current. contains (event. target)) {

          return;

        }

        setOptionVisible(false);

      },

      {capture: true}

    );

  }, []);

If (**ref. current)** contains (**event. target)** then we return nothing or it means our addEventListener will do nothing otherwise it will change our state to false (eventually closing down our dropdown).

**Note:**

* contains method belongs to all DOM elements and allows us to check whether one DOM element is inside another DOM element.
* Leaving return statement blank in our case means we return early so that we don’t attempt to setOptionVisible to false

The Next question is about refactoring our current code implementation to avoid future possible bugs. So please pay close attention on that as well.

Q> **Why we need to add a cleanup function in the useEffect that we used for closing our dropdown when we click outside its scope and how we will achieve this**?

A>

1. Suppose we remove Dropdown from our App component using a button to toggle between showing and removing our Dropdown component.

const [showDropdown, setShowDropdown] = useState(true);

<React.Fragment>

      <button onClick= {() => setShowDropdown (! showDropdown)}>

        Toggle Dropdown

      </button>

      {showDropdown? (

        <DropdownExample

          colorName={colorName}

          setColorName={setColorName}

          selected={selected}

          setSelected={setSelected}

          options={options}

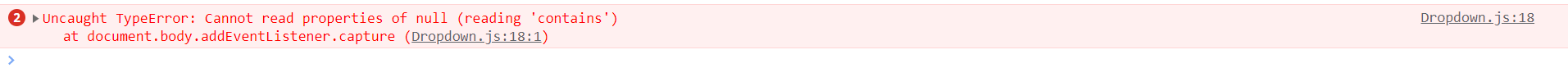
        />

      ): null}

    </React.Fragment>

When showDropdown is false we return null in other words our Dropdown component gets removed from DOM.

Now if we click on anywhere on the screen, we get this error in console.



This error is coming since our ref used in useRef hook is returning null now. Since browser can’t read properties of null and therefore we get this error.

Now in order to solve this problem, we need to make sure that whenever our Dropdown component is removed from DOM, we need to turn off its document. body event listener

So in order to do that we will use cleanup function of our useEffect hook. Just return a function saying document. body. **removeEventListener** which will be invoked in case we remove our Dropdown component.



Note: *If useEffect gets called twice for no reason, maybe it’s happening due to <React.Strictmode>*

**Q> Translate widget Component scaffolding**

Components/HTML Elements/Headings & Label/Array of Objects/State objects

**Component #1 Translate**

**Purpose:**

1. Show Input with label “Enter Text”.
2. Show Dropdown with label “Select Language”.
3. Show Convert component.

**Component #2 Dropdown**

**Purpose:**

1. Show Label “Select Language”.
2. Show all languages as items in Dropdown form
3. User can select a particular language item.

**Component #3 Convert**

**Purpose:**

1. Show a Label “Output”.
2. Show Translated Text as a h1 heading in bold.

Q> **What does window. location object gives us and explain its pathname property with example**?

A>

1. This window. location is an object that is built into your browser.
2. Any time we navigate around to a different URL, it gets updated.
3. Inside this object, we can see a lot of information which is extracted from the current URL like current hostname, the entire URL, origin, pathname, port, protocol etc…
4. We are generally more focused on **pathname** property which is everything inside the URL after the domain and port.
5. Whenever we are at our root route, we see our pathname as **“/”** *a forward slash*. In other words,

**window. location. pathname** === “/”.

const showAccordion = () => {

  if (window. location. pathname === "/")

return <Accordionitems={items} />;

};

const App = () => {

  return (

    <React.Fragment>

      {showAccordion ()}

    </React.Fragment>

  );

};

export default App;

Here we can see, how we can use this property to help us show components based on current pathname

Q> **How can we create a reusable React Route component?**

A> **Step 1**: We will do ***rough Route Mapping*** which is to show particular component on the screen based upon specific URL

e.g. locahost:3000 🡪Accordion

localhost:3000/list 🡪Search

localhost:3000/dropdown 🡪 Dropdown

localhost:3000/translate 🡪 Translate

**Step 2:** ***Implement Route map logic* in** our app using **window. location. pathname** property.

const showAccordion = () => {

  if (window. location. pathname === "/")

return <Accordion items={items} />;

};

const App = () => {

  return (

    <React.Fragment>

      {showAccordion ()}

    </React.Fragment>

  );

};

export default App;

Now, we are only showing Accordion component when our current URL shows locahost:3000. For other components, you can observe the same behavior by giving appropriate pathname and also some repetitive code.

**Step 3:** **Create a Route component,** which takes in two props path (the URL path) and children (component that you want to render on that path).

const Route = ({path, children}) => {

  return window. location. pathname === path? children: null;

};

export default Route;

**Step 4:** In App.js **import Route component and provide its props proper values**.

      <Route path="/">{<Accordion items={items} />}</Route>

      <Route path="/translate">{<Translate />}</Route>

      <Route path="/list">{<Search />}</Route>

      <Route path="/dropdown">

        {

          <DropdownExample

            label="Select a Color"

            colorName={colorName}

            setColorName={setColorName}

            selected={selected}

            setSelected={setSelected}

            options={options}

          />

        }

      </Route>

Here we have provided our components as children prop to Route component. We can insert this Route component anywhere into our application and do a lot of interesting stuff.

**Step 5:** **Create a Header/Navbar component**

const Header = () => {

  return (

    <div className="ui secondary pointing menu">

      <a href="/" className="item">

        Accordion

      </a>

      <a href="/list" className="item">

        Search

      </a>

      <a href="/dropdown" className="item">

        Dropdown

      </a>

      <a href="/translate" className="item">

        Translate

      </a>

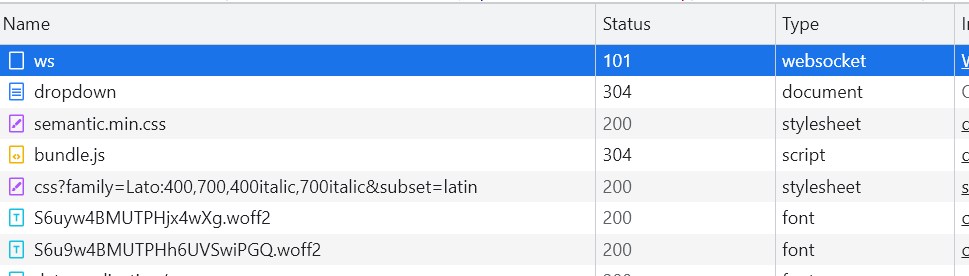
    </div>

  );

};

We are using anchor tag with href property pointing to pathname and when we click on any anchor tag, our URL pathname changes and webpage renders the relative component.

But there is a **big issue** in this approach.



Above are all the network requests made whenever we click on any of the anchor tags, it’s not ideal for web applications.

It reloads our page on every click with index.html which contains all our css and JavaScript files. In react we don’t need our app to hard reload every time in order to render our assets.

**Step 6:** **Create a Link component and preventReload.**

The purpose of this component is, whenever a user clicks on a link, make the URL and all our routes updated but not refresh the page.

It will just show a Link or a normal anchor element on the screen where we will attach an onClick handler that is going to execute a logic whenever user clicks on the element.

const Link = ({href, children, className}) => {

  const preventReload = (event) => {

    event. preventDefault ();

  };

  return (

    <a className={className} href={href} onClick={preventReload}>

      {children}

    </a>

  );

};

After clicking on element, we can see a navigation event is triggered whose job is to let entire application know that our URL has changed.

This navigation event will be sent to all of the Route component and they will decide whether or not to show their respective components / children.

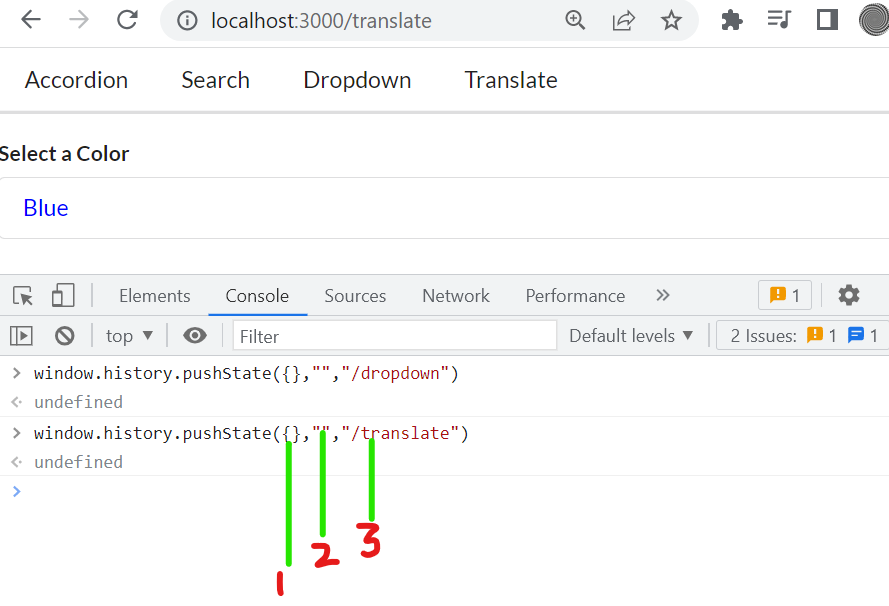
**event. preventReload** 🡪Prevent a full page reload after onClick event.

**Step 7:** **Change the URL**

To understand this step we first have to understand why and how we are doing this.

Whenever a user navigates to some page, they generally have the expectation that they should be able to bookmark current page and then come back in future and see the same content as before.

That’s why as developer it is paramount that URL is kept in sync with the content that is displayed on the screen.



window. history. pushState method is used for this purpose

Its *first argument is an empty object*.

*Second argument is empty string*.

*Third argument is the pathname*.

When you pass these arguments in console, you can see current URL changing as per pathname provided in pushState method.

const Link = ({href, children, className}) => {

  const preventReload = (event) => {

    event. preventDefault ();

    window. history. pushState ({}, "", href);

  };

  return (

    <a className={className} href={href} onClick={preventReload}>

      {children}

    </a>

  );

};

Now when you click on the Links you will see URL being updated.

**Step 8:** **Produce and emit navigation event to all Route components.**

const Link = ({href, children, className}) => {

  const preventReload = (event) => {

    event. preventDefault ();

    window. history. pushState ({}, "", href);

    const navigationEvent = new PopStateEvent("popstate");

    window. dispatchEvent (navigationEvent);

  };

  return (

    <a className={className} href={href} onClick={preventReload}>

      {children}

    </a>

  );

Purpose of these two lines is just to communicate to Route component that URL has changed

    const navigationEvent = new PopStateEvent("popstate");

    window. dispatchEvent (navigationEvent);

**Step 9:** **Listening to navigation event inside Route**

We will implement useEffect with empty array as dependency

const Route = ({path, children}) => {

  useEffect (() => {

    const onLocationChange = () => {

      console.log ("Location change");

    };

    window. addEventListener ("popstate", onLocationChange);

    return () => {

      window. removeEventListener ("popstate", onLocationChange);

    };

  }, []);

  return window. location. pathname === path? children: null;

};

export default Route;

As you can see we are executing onLocationChange function from console, when Route detects popstate event.

**Step 10:** **Update/Re-render Route component whenever URL is updated**.

Right now our Route is rendering its child based on window. location. pathname so in order to re-render our Route component let ‘s use useState hook.

const Route = ({path, children}) => {

  const [currentPath, setCurrentPath] = useState (window. location. pathname);

  useEffect (() => {

    const onLocationChange = () => {

      setCurrentPath (window. location. pathname);

    };

    window. addEventListener ("popstate", onLocationChange);

    return () => {

      window. removeEventListener ("popstate", onLocationChange);

    };

  }, []);

  return currentPath === path? children: null;

};

**Additional Refactoring:**

If we press ctrl and then click on some link, it opens the Link in a new tab. Right now our App does not do that.

  const preventReload = (event) => {

    if (event. metaKey || event. ctrlKey) {

      return;

    }

metaKey is for mac OS and ctrlKey is for windows OS based systems.

So in these 10 steps, we can create some basic navigation without making use of react router or anything like that.

Q> **What are the basics of converting a Class based react component into a functional component**?

A>

1. The state object of class component must be transformed for useState hook.
2. All event handler functions must have proper syntax (use const keyword before each of them).
3. Instead of setState function *use the useState’s setter function to update state value*.
4. Make sure there are *no naming conflicts of event handler functions with props* provided to our component.
5. Remove **this. state** keyword from our component whenever we reference our state object.
6. Copy all the JSX from return function of render method and paste it as it is inside our functional component.
7. Now remove **this** and **this. state** keywords from wherever we have mentioned it in our JSX.

**Searchbar as a Class Component**

class Searchbar extends React.Component {

  state = {term: "“};

  handleInputChange = (e) => {

    this. setState ({term: e. target. value});

  };

  handleFormSubmit = (e) => {

    e. preventDefault ();

    return this. props. onSearchBarSubmit (this. state. term);

  };

  render () {

    return (

      <div className="search-bar ui segment">

        <form onSubmit={this. handleFormSubmit} className="ui form">

          <div className="field">

            <label>Video Search</label>

            <input

              type="text"

              onChange= {this. handleInputChange}

              value= {this. state. term}

            />

          </div>

        </form>

      </div>

    );

  }

}

export default Searchbar;

**Searchbar as functional component after refactoring**:

const Searchbar = ({onSearchBarSubmit}) => {

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

  const handleFormSubmit = (e) => {

    e.preventDefault();

    return onSearchBarSubmit(term);

  };

  return (

    <div className="search-bar ui segment">

      <form onSubmit={handleFormSubmit} className="ui form">

        <div className="field">

          <label>Video Search</label>

          <input

            type="text"

            onChange={(e) => setTerm (e. target. value)}

            value={term}

          />

        </div>

      </form>

    </div>

  );

};

*Note that, Code seems small after refactoring*.

Q> **What is Redux**?

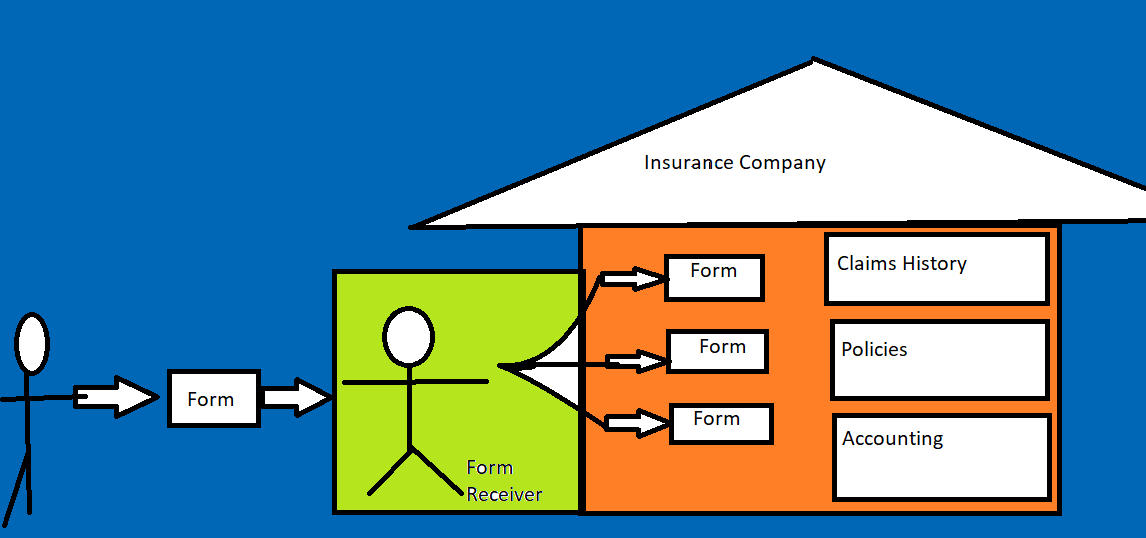
A>

1. It is a state management library, so rather than authoring state or maintaining state inside of our React components, we can extract it to our Redux library.
2. As compared to React library itself which is all about rendering content on the screen and handling user interaction, its primary focus is around maintaining, updating or handling data.
3. It makes creating complex application easier.
4. Redux Library is not explicitly designed to work with React. It can work with other languages and libraries as well.

Q> **Explain Redux with the help of an analogy**?

A>

1. The best analogy we can give is of a *typical insurance company*, which requires common people to pay some amount of money every month and in case something bad happens like a car crash or some accident, they get paid some predefined lump sum amount from the insurance company.
2. Now two terms are very commonly used by insurance companies, **policy** and **claim**.
3. **Policy**: A customer purchase or hold a policy. It is a *contract* between customer and insurance company which basically says if something bad happens to the customer, the insurance company will pay some X amount of money.
4. **Claim**: The request which customer files or reports to an insurance company in order to get some money as per the policy.
5. Now with these terms in mind, lets dig deep into Customer **🡪 🡨**Insurance company interaction. Diagram below will help you better understand this.



Three departments in insurance company are:

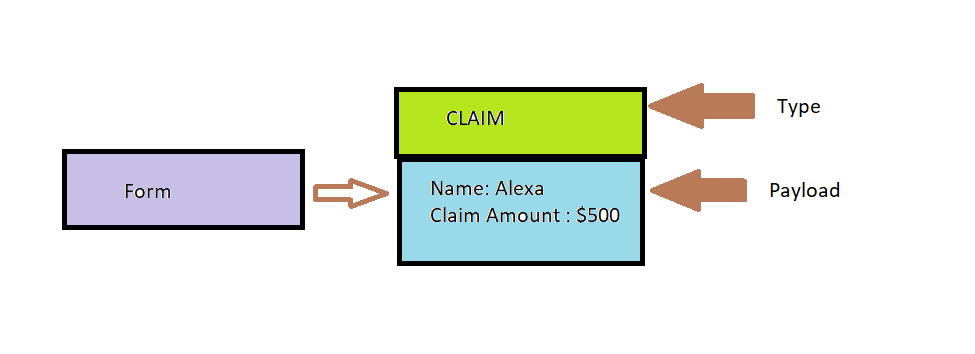
1. **Claims History**: Stores a list of all claims ever made.
2. **Polices**: Stores a list of people who has policy with insurance company.
3. **Accounting**: Stores a big bag of cash we pay people from this.

Now, Let’s understand this diagram from left to right: -

1. In order for our customer to set up a policy with our insurance company, he/she will fill out a form and visit to the company headquarter to submit it.
2. Now, customers can’t directly go to one of these company departments to submit form instead they are going to hand it off to some front office, that we will refer to as the **form receiver** whose *main job is to take the form and make copies of it*.
3. *Copies of form are provided to each of the department*. Even if they don’t need it.
4. The process is quite simple and straightforward up to this point, so now let’s add some complexity to dive deep into this Redux analogy.
5. When customer gave a form to sign up for a new policy, the only relevant department will be policies department whose main purpose is to *store a list of users with active policy*.
6. Let’s imagine we have a **management team** who comes knocking on the door of policies department and ask to give a list of active policy users for maybe some metric for marketing reason. Nonetheless, they get tired eventually of this process of involving policies department every time they need some data related to active users.
7. The workaround for this problem is to *store policies department data inside of a* ***central repository*** (and eventually other department’s data as well). So now the management team can very easily get any data they want.
8. It is the job of form receiver now, to make one more copy of the form and give it to central repository.
9. So policies department will get a list of all our current policies from central repository along with copy of newly added user from form receiver. It will update the list of policies and then pass updated list back to central repository.

Get List and Form **🡪** Update list with new form**🡪**Pass updated list to central repository.

1. Now let’s focus on the Form itself. Suppose every form which is filled by the customer has two different fields. A *type* and a *Payload*.
2. The type tells, what’s the purpose of the form is. Now for insurance company *it can be either a* ***policy*** *that customer can create* or they *can make a* ***claim*** over existing policy.
3. This means we will have different forms for creating a policy or claim. Here purpose of the form can be defined as type on the very top and inside the body of the form which we call as payload customer will write some information relevant to them.
4. In case customer create a claim type of form, we might need their name and the claim amount they require from the insurance company.



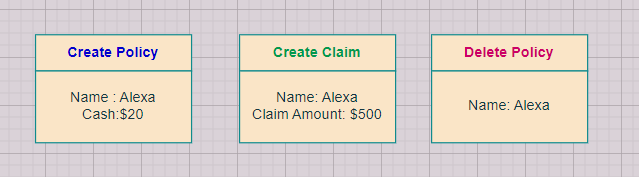
1. In Total, we might need three different types of Form

🡪Create Policy

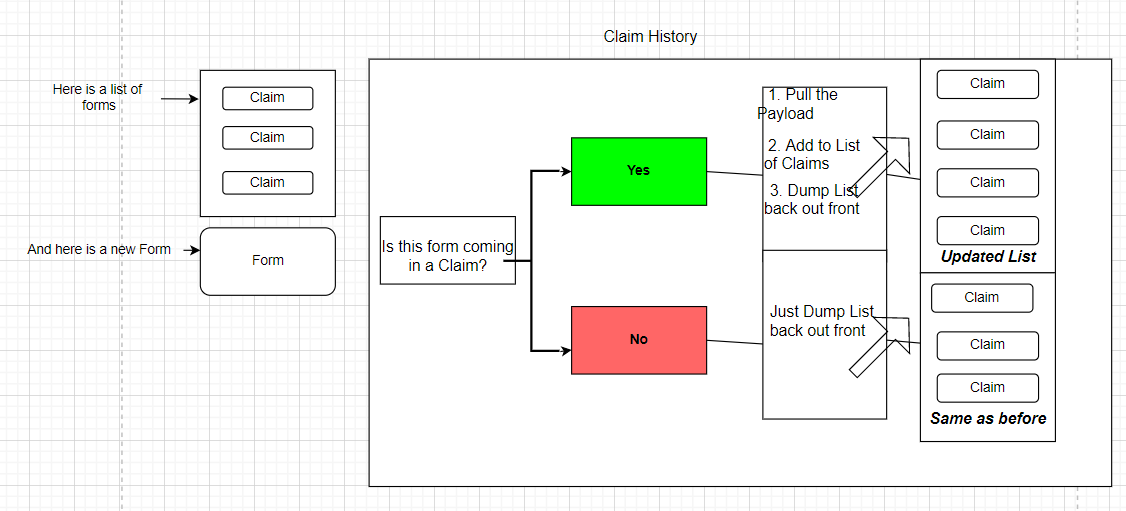
🡪Create Claim

🡪Delete Policy

Each of these type of form will have different set of information inside its payload.

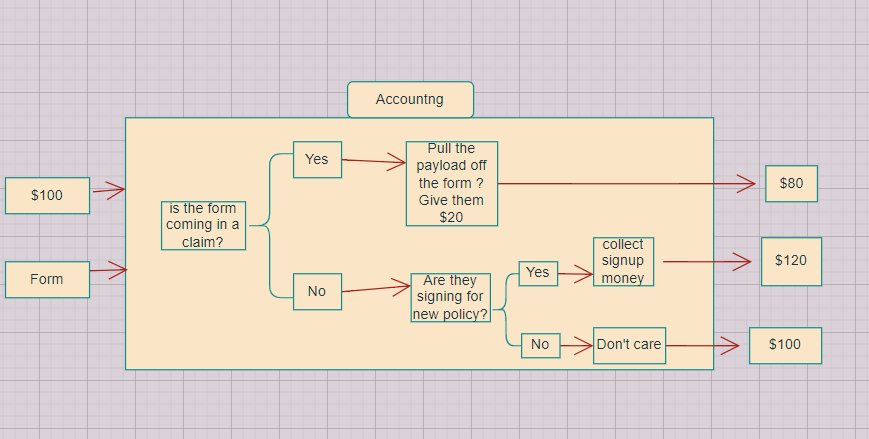


**Types of form with their payload**



Above is **Claim History department**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**



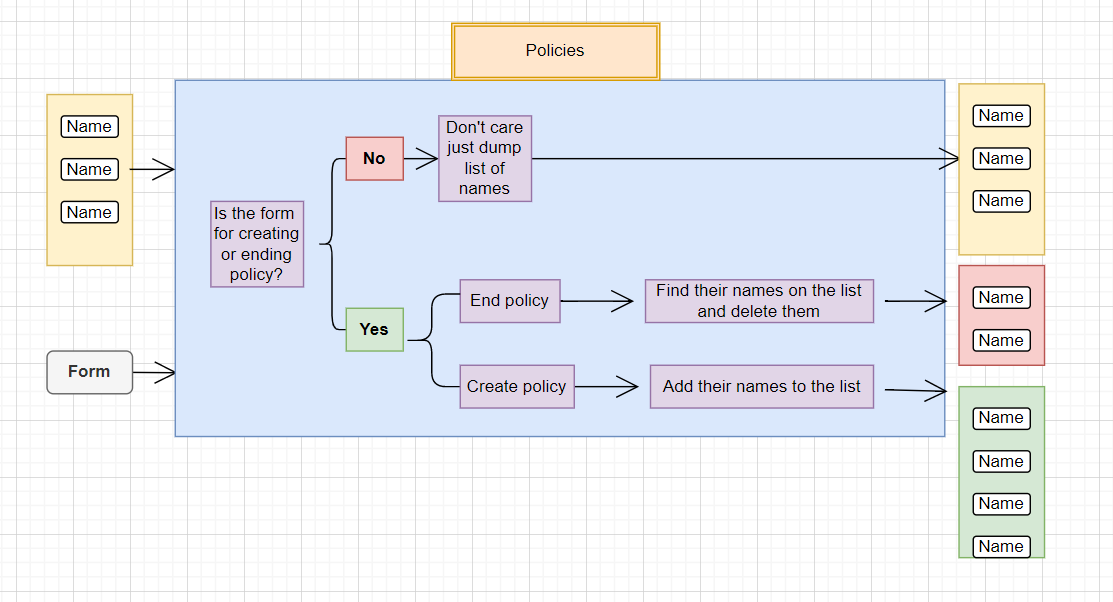
Above is **Accounting department**

Accounting department will have a bag of money let’s say $100 which is current company balance and a Form that is received from Form receiver.

If it’s a claim form for $20, they will give it to customer and keep remainder $80.

On the other hand, if the form they received is a type of create policy, the cash required for signup will be collected and balance will be updated accordingly

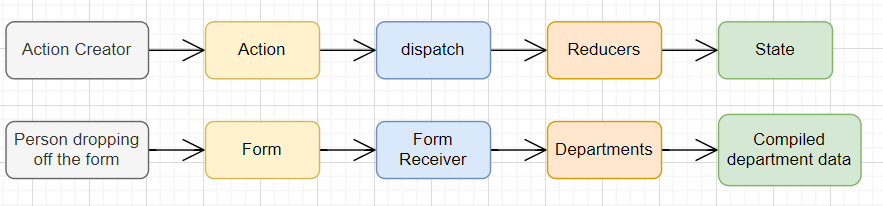
If it’s neither same balance will be returned as before to the central repository.

****

**Above is the policies department**

*Most important thing we have to notice is that how all the individual department’s data is stored in one central repository which gets updated every time a new Form is received*.

Now we will see how this analogy of insurance company and its departments fit precisely into our understanding of Redux.



Redux Cycle **🡨🡪**Insurance company

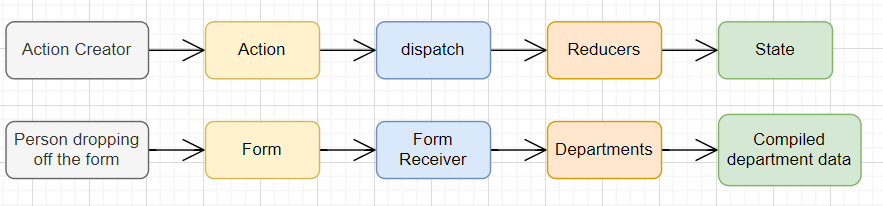
**In Insurance company** 🡪

1. Everything in insurance company starts with a **person dropping the form** off. 🡪**Action Creator**
2. That **form** contains some amount of information *a type and some payload* on how they want to change some data within our company. 🡪**Action**
3. We would take that form, make copies of it and pass it off to each different department, and then it would be up to our department to look at the details on the form and decide how to change the data that our insurance company is currently maintaining. 🡪**Action**
4. **Form receiver** makes the copies of the form and hand it off to each department 🡪**Dispatch**
5. Each **Department** essentially has its own little separate slice of data like a list of policies/list of claims/current balance🡪**Reducers**.
6. When the department is done processing the data, it spat out some list of data into **compiled department data** 🡪**State**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**In Redux** 🡪

1. **Action Creator** is a function that is going to create or return a plain JavaScript object.
2. This plain JavaScript object is called an **Action**. This action has a type property and a payload property
3. The type property on an *action describes some change that we might want to make inside of our data* and then the payload property *describes some context around the change that we want to make*.
4. **Purpose of an action** is to describe some change that we want to make to data inside of our application.
5. **Dispatch function** is going *to take the action, make its copies of action object and then pass it off to a bunch of different places inside our application*.
6. **Reducer function** is *responsible for taking an action and some existing slice of data*. It will *process that action* and then *make some change to the data* and *finally return it* so that it can be used in some centralized location. It’s going to take in an action, look at the action, look at its type then based on that type decide how to update its data.
7. **State object** in Redux is a *central repository of all information that has been created by our reducers*.
8. All the information gets consolidated inside our state object so that our React application can very easily reach into our Redux side of the app and get access to all of the data inside our application.
9. In that way, our *React app doesn’t have to go around to each separate reducer and get current list of policies/current state so to speak*.



Now we will see the first two steps in action together *Action Creator and Action*.

Just remember that action creator is just a function that is going to return a plain JavaScript object referred to as the action.

The action is like our form which will have a type that describes the purpose of our form and a payload that provides some context on exactly what this form is doing.

We are first going to create an action creator function createpolicy which will take name and amount of money from customer.

const createPolicy = (name, amount) => {

};

For each type of action, we will create a separate action creator.

console. clear ();

//People dropping off the form --> Action Creator

const createPolicy = (name, amount) => {

 return {//Action for create policy form

  type:"CREATE\_POLICY",

   payload: {

     name,

     amount

   }

 };

};

const deletePolicy=(name)=> {

  return {//Action to delete policy form

    type:"DELETE\_POLICY",

    payload: {

      name

}

  }

}

const createClaim = (name, amountOfMoneyToCollect) => {

 return {//Action to create claim form

  type:"CREATE\_CLAIM",

   payload: {

     name, amountOfMoneyToCollect

   }

 };

};

As you can see each action creator function returns a JavaScript object with type and payload property or in Redux terminology an Action.

Notice value of type is in all **CAPS**; *this is the general convention in Redux Action object*.

Now we will **skip the dispatch** step of the redux cycle. Remember that dispatch is like our form receiver, who is going to receive a form or an action, make a copy of it and send it out to all the different departments.

*This dispatch is actually a part of the redux library itself, so we don’t have to write it from scratch, so we will skip dispatch and move directly to implement reducers*.

For **reducers**we will write a couple of function and each function is going to model a different department inside of our company.

For example, first reducer is in charge of modeling behavior of our **claims history** department, then second reducer for **accounting department** and third reducer for our **policies department**.

Each of these *reducers will be called within action* that was previously created by action creator.

The *reducer is then going to inspect that action and decide whether or not it needs to modify some data based upon that action*.

Remember, we are doing this entire process so that all the data associated with every department can be monitored from a single location / central repository.

And only when we have a form to pass in a department would we take that department’s slice of data from central repository and pass it into that department. This will become relevant when we put together this reducer. This below is a general reducer function

const claimsHistory = (oldListOfClaims, action) => {

  if (action. type === "CREATE\_CLAIM") {

    //we care about the action

  }

//we don't care about the action

 }

This function always gets two arguments that are always passed in the exact same order.

The *first argument is going to be whatever existing little piece of data from our central repository that belongs to this particular department/reducer*.

We are calling this slice of data as oldListOfClaims in our case where this reducer represents our claimsHistory department. *The idea behind this is that reducer must be able to update this list of claims depending upon the contents of an action which is passed as the second argument*.

Look at the diagram workflow for claims history department and you will see that If the action is of type CREATE\_CLAIM, we will get the payload of that action and add it into our oldListOfClaims.

const claimsHistory= (oldListOfClaims= [], action) => {

  if (action. type === "CREATE\_CLAIM") {

    //we care about the action

    return [...oldListOfClaims, action. payload]

  }

  //we don't care about the action

  return oldListOfClaims;

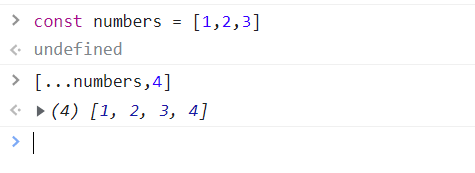
}

This part we are returning

return [...oldListOfClaims, action. payload]

let’s focus on this first. It is ES2015 syntax of creating a new array after adding a new element from action. payload.

You can use this example to understand this syntax:



*This syntax always creates a brand new array instead of push method* ***(oldListOfClaims.push(action. payload)****) which modifies the original array*.

This is done to avoid modifying existing data structures inside of reducer as much as possible.

**Important rule around reducer:** *This reducer has no idea what its data is until it gets called for the first time. So in case there is no data inside oldListOfClaims, we will receive the value of undefined as first argument. So we will default it just an empty array*.

**const claimsHistory= (oldListOfClaims= [], action)**

Next two reducers are based on the same principle as well

const accounting= (bagOfMoney=100, action) => {

  if (action. type === "CREATE\_CLAIM") {

    return bagOfMoney - action. payload. amountOfMoneyToCollect

  }

  else if (action. type === "CREATE\_POLICY") {

    return bagOfMoney + action. payload. amount

  }

  return bagOfMoney;

}

const policies= (listOfPolicies= [], action) => {

 if (action. type === "CREATE\_POLICY") {

    return [...listOfPolicies, action. payload. name]

  }

  else if (action. type === "DELETE\_POLICY") {

    return listOfPolicies. filter (name => name! == action.payload.name)

  }

  return listOfPolicies

}

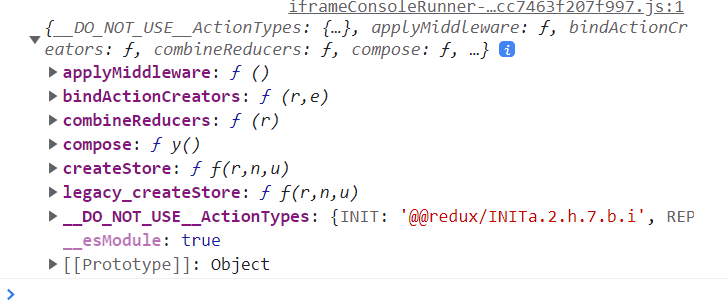
**Note:** **Overall goal of reducer is to take some existing data, some action and then modify and return that existing data based upon the contents of an action**.

Now all our baseline stuff which is action creator, action and reducers are put together, now we will create a new instance of something called Redux store.

Q> What exactly is Redux store and how it’s used along with actions and reducers?

A>

1. Vast amount of code that we write with Redux is just plain and simple functions/arrays/objects and only at very specific points we reference the Redux Library directly.
2. Since we have put all our action creators and reducers together, now we are going to wire them into a single object called store.
3. A store in redux is essentially the assembly of a collection of different reducers and action creators.
4. After importing Redux into your project, if you console.log(Redux), you will see Redux library in the console.



1. We will take two specific functions from this library

const {createStore, combineReducers} = Redux;

1. To wire all our reducers together, we will use **combineReducers** function.

const ourDepartments = combineReducers ({//all our reducers that we put together inside object as key: value pairs

  accounting: accounting,

  claimsHistory: claimsHistory,

  policies: policies

})

All the reducers are now combined into a single instance called ourDepartments of combineReducers function, which takes an object as its argument.

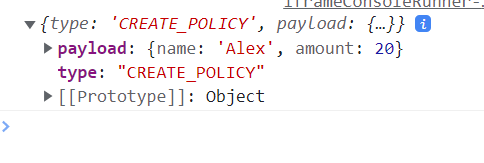
1. Now we will create an *object* called store from **createStore** function in which we will pass ourDepartments.

const store = createStore(ourDepartments);

1. **This store object represents our entire redux application***. It contains reference to all our different reducers and to all the data produced by those reducers as well*. This store object has a lot of important methods that we can use.
2. One of the methods is the **dispatch** method. Very much like the form receiver in our insurance company analogy. We pass this dispatch method an action *object* so that it can make a copy of it and send it off to all the reducers.
3. Now in order to create an action we will call one of our action creator function called createPolicy.

const action = createPolicy("Alex",20); 🡨Remember it is just a function so pass in the valid values as argument.

If you console.log(action), you will see

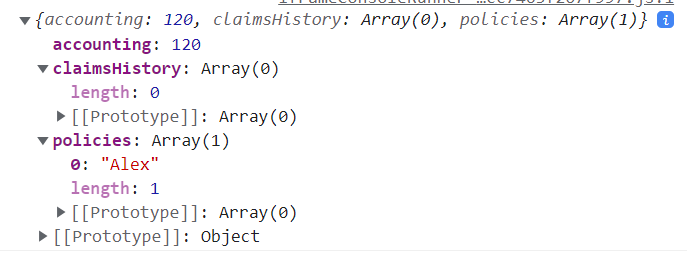


1. Now this action object is passed into store. dispatch.

**store. dispatch**(action);

Once we pass the action into store. dispatch, this action will reach out to each and every reducer inside our store.

1. To see it console.log (store. getState ())



**getState** is a function that gets information of all the repository of data used in our application inside an object. So when we get this state object, we get access to this gigantic blob of information.

1. So now we can call all of our different action creators, pass the action that gets returned to store. dispatch and that in turn modify our application data based on our reducers logic.

store. dispatch(createPolicy("jim",50));

store. dispatch(createPolicy("Alex",20));

store. dispatch(createPolicy("bob",20));

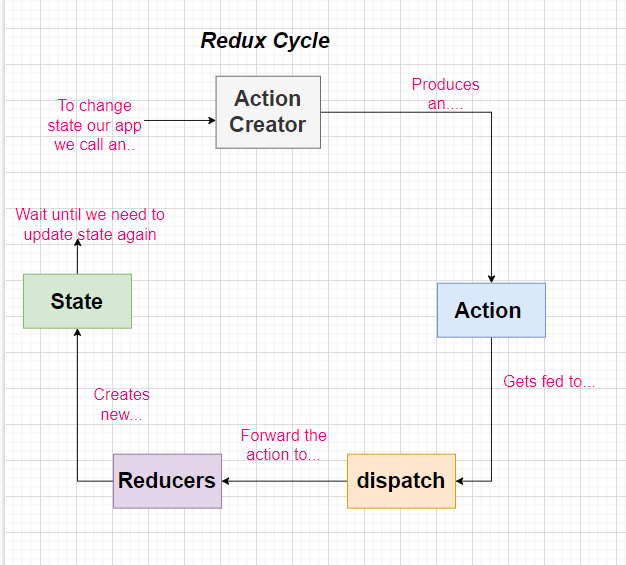
store. dispatch(createClaim("Alex",20));

store. dispatch(deletePolicy("bob"));

You can see three new policies are made for Jim, Alex and Bob.

A claim is made for $20 by Alex and bob decided to delete his policy. If you console log store. getState you will see that all these actions make simultaneous changes in the store’s state.\

Q> **Explain Redux cycle with a diagram**?



**Note**: All these dispatchers below start a new and individual redux cycle when they are called.

store. dispatch(createPolicy("jim",50));

store. dispatch(createPolicy("Alex",20));

store. dispatch(createPolicy("bob",20));

store. dispatch(createClaim("Alex",20));

**Note:** We can’t modify the state using some algebraic expression directly. The only way we can change state is by passing action object into store. dispatch.

Q> **Why you use Redux in your application**?

A>  
As we start working on an application, it gets more and more complicated with time as there is much more data that needs to be handled eventually.

But with Redux we get more stable time-complexity curve, because our application automatically documents its data using Redux store.

It is also less prone to code complexity and programming errors for new developers who start working on my code because data can only be changed using specific action-creators and reducers, so it’s kind of pre-defined that how data is being handled by application.

Q>**When does this Uncaught Error” Minified Redux error #12 appears**?



In Redux official documentation site error code #12 is depicted as

The slice reducer for key "" returned undefined during initialization. If the state passed to the reducer is undefined, you must explicitly return the initial state. The initial state may not be undefined. If you don't want to set a value for this reducer, you can use null instead of undefined.

I observed this error when I did not initialize piece of original data which is the first parameter of my reducer function

const policies= (listOfPolicies, action) => {

 if (action. type === "CREATE\_POLICY") {

    return [...listOfPolicies, action. payload. name]

  }

  else if (action. type === "DELETE\_POLICY") {

    return listOfPolicies. filter (name => name! == action.payload.name)

  }

  return listOfPolicies

}

Notice that listofPolicies is not initialized to anything, so it returns undefined

After just initializing it to an empty array, its solved

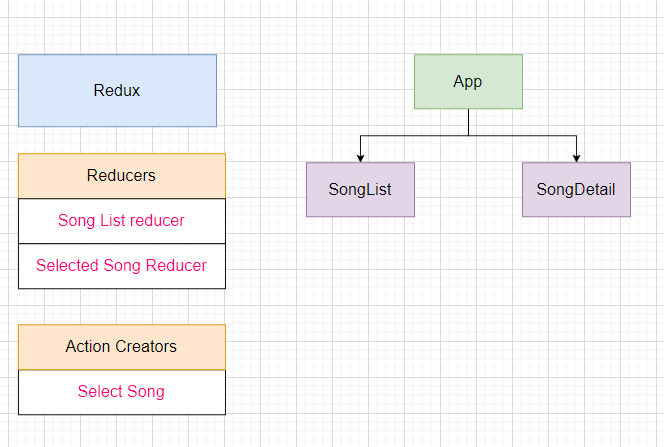
const policies= (listOfPolicies= [], action) => {

Q> **How to use redux inside React**?

1. We are going to install redux for redux library functions alongside react-redux so that react and redux can communicate to one another.
2. npm install --save redux react-redux.
3. Since redux was not originally designed to work only with react, therefore react-redux library has a bunch of helper functions inside of it to get Redux to work nicely with react.

Q> **Create an app for selecting a song from the list of songs and displaying its information using Redux**?

A>



1. We will use three components in total our main component App, SongList and Songdetail. App component will be passing very little information down to its child component.
2. We are going to abstract out all the functionalities like *creating a list of songs*, *selecting a song*, and *what the currently selected song is* into the redux side of application.
3. A reducer will produce the list of songs another reducer records what currently selected song is that’s *2 pieces of state inside our application*.
4. We have to make sure we have an action creator in order to change our current states. Remember, this is the only way to change any state inside a Redux application.
5. Action creator can be called something like SelectSong that will dispatch an action and tell this selected song reducer to update its data and reflect the newly picked up selected song.
6. Song List Reducer is going to pass a static list of songs like a fixed array of objects.

Q> **How React-Redux library helps connecting Redux application with React application**?

A>

1. Using React-Redux we are going to create two new instances for components called Provider and Connect.
2. We will pass some props into both of them to configure exactly how they work.

**Follow up on your reads:**

Q> **How to use code pen and add dependencies like redux?**

A> **Refer to Lecture 224 on udemy**

Q> **Create a reusable Dropdown component so that your main App component keep track of it and manage its state accordingly**.

1. **Refer to Dropdown.js**

**Q>What is useCapture parameter in addEventListener?**

<https://stackoverflow.com/questions/7398290/unable-to-understand-usecapture-parameter-in-addeventlistener>

**Q> Detect Click outside React component?**

<https://www.geeksforgeeks.org/how-to-detect-click-outside-react-component/#:~:text=We%20can%20use%20the%20createRef,a%20reference%20for%20any%20element>.

**Q> Customize a reusable React dropdown menu component?**

<https://blog.logrocket.com/customize-reusable-react-dropdown-menu-component/>

**Q> Understanding React’s useEffect cleanup function?**

<https://blog.logrocket.com/understanding-react-useeffect-cleanup-function/>

**Q> How to manage useEffect’s dependency array like a Pro?**

<https://www.bam.tech/article/how-to-avoid-bugs-in-useeffect#:~:text=The%20useEffect%20hook%20allows%20you,of%20the%20useEffect%20has%20changed>.

**Q> A beginner’s guide to use Redux with React?**

<https://bretcameron.medium.com/a-beginners-guide-to-redux-with-react-50309ae09a14>

**Q> Build a password manager in React / MySQL / Node**

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