# 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.
3. (position) =>
4. this.setState({
5. lat: position.coords.latitude,
6. long: position.coords.longitude,
7. })

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?**

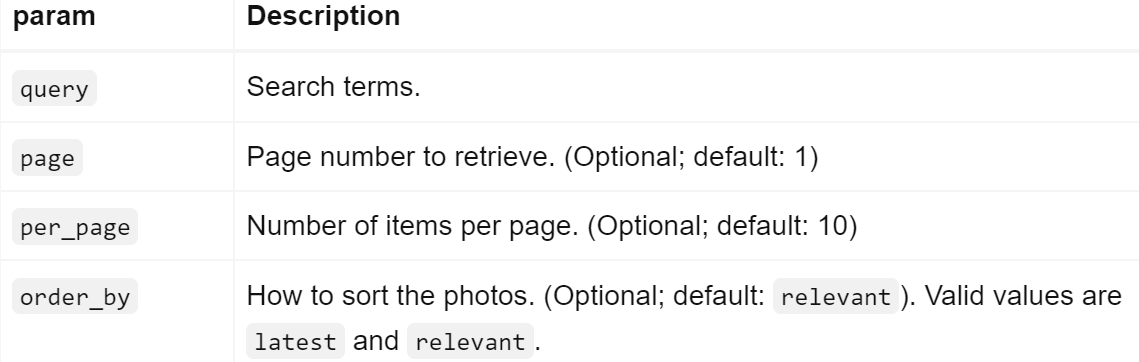
A>  
To make a network request to a third party API, It’s not going to be job of the React library itself.

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) => {

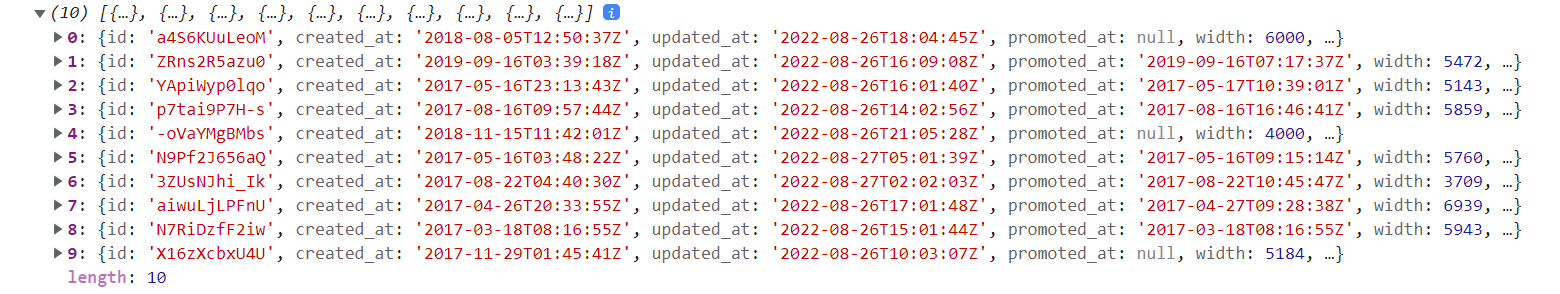
  console.log (props. imageprop);

  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.