UNIT IV Advanced Client side Programming

**Server-side Programming :**

It is the program that runs on server dealing with the generation of content of web page.  
1) Querying the database  
2) Operations over databases  
3) Access/Write a file on server.  
4) Interact with other servers.  
5) Structure web applications.  
6) Process user input. For example if user input is a text in search box, run a search algorithm on data stored on server and send the results.

Examples :  
The Programming languages for server-side programming are :  
1) PHP  
2) C++  
3) Java and JSP  
4) Python  
5) Ruby on Rails

**Client-side Programming :**

It is the program that runs on the client machine (browser) and deals with the user interface/display and any other processing that can happen on client machine like reading/writing cookies.

1) Interact with temporary storage  
2) Make interactive web pages  
3) Interact with local storage  
4) Sending request for data to server  
5) Send request to server  
6) work as an interface between server and user

The Programming languages for client-side programming are :  
1) Javascript  
2) VBScript  
3) HTML  
4) CSS  
5) AJAX

[ReactJS](https://www.javatpoint.com/reactjs-tutorial)

[ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components. It is an open-source, component-based front end library responsible only for the view layer of the application. It was created by](https://www.javatpoint.com/reactjs-tutorial)**[Jordan Walke,](https://www.javatpoint.com/reactjs-tutorial)**[who was a software engineer at](https://www.javatpoint.com/reactjs-tutorial)**[Facebook.](https://www.javatpoint.com/reactjs-tutorial)**[It was initially developed and maintained by Facebook and was later used in its products like](https://www.javatpoint.com/reactjs-tutorial)**[WhatsApp](https://www.javatpoint.com/reactjs-tutorial)**[&](https://www.javatpoint.com/reactjs-tutorial)**[Instagram.](https://www.javatpoint.com/reactjs-tutorial)**[Facebook developed ReactJS in](https://www.javatpoint.com/reactjs-tutorial)**[2011](https://www.javatpoint.com/reactjs-tutorial)**[in its newsfeed section, but it was released to the public in the month of](https://www.javatpoint.com/reactjs-tutorial)**[May 2013.](https://www.javatpoint.com/reactjs-tutorial)**

[Today, most of the websites are built using MVC (model view controller) architecture. In MVC architecture, React is the 'V' which stands for view, whereas the architecture is provided by the Redux or Flux.](https://www.javatpoint.com/reactjs-tutorial)

[A ReactJS application is made up of multiple components, each component responsible for outputting a small, reusable piece of HTML code. The components are the heart of all React applications. These Components can be nested with other components to allow complex applications to be built of simple building blocks. ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.](https://www.javatpoint.com/reactjs-tutorial)

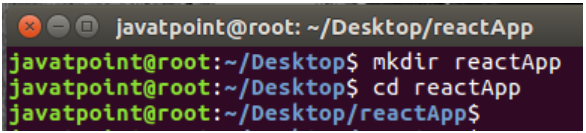
[To create React app, we write React components that correspond to various elements. We organize these components inside higher level components which define the application structure. For example, we take a form that consists of many elements like input fields, labels, or buttons. We can write each element of the form as React components, and then we combine it into a higher-level component, i.e., the form component itself. The form components would specify the structure of the form along with elements inside of it.](https://www.javatpoint.com/reactjs-tutorial)

ReactJS is a **declarative**, **efficient**, and flexible **JavaScript library** for building reusable UI components. It is an open-source, component-based front end library which is responsible only for the view layer of the application. It was initially developed and maintained by Facebook and later used in its products like WhatsApp & Instagram.

The main objective of ReactJS is to develop User Interfaces (UI) that improves the speed of the apps. It uses virtual DOM (JavaScript object), which improves the performance of the app. The JavaScript virtual DOM is faster than the regular DOM. We can use ReactJS on the client and server-side as well as with other frameworks. It uses component and data patterns that improve readability and helps to maintain larger apps.

**Install React and React DOM**

Create a **root** folder with the name **reactApp** on the desktop or where you want. Here, we create it on the desktop. You can create the folder directly or using the command given below.



# ReactDOM

The react-dom package provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside the React model if you need to.

import \* as ReactDOM from 'react-dom';

In the previous article on [introduction to JSX](https://www.geeksforgeeks.org/reactjs-introduction-jsx/), we learned that we can use JSX to store HTML markups within Javascript variables. Now, ReactJS is a library to build active User Interfaces thus rendering is one of the integral parts of ReactJS. React provides the developers with a package **react-dom** a.k.a ReactDOM to access and modify the DOM. Let’s see in brief what is the need of having the package.

**What is DOM?**

DOM, abbreviated as Document Object Model, is a World Wide Web Consortium standard logical representation of any webpage. In easier words, DOM is a tree-like structure that contains all the elements and it’s properties of a website as its nodes. DOM provides a language-neutral interface that allows accessing and updating of the content of any element of a webpage.

Before React, Developers directly manipulated the DOM elements which resulted in frequent DOM manipulation, and each time an update was made the browser had to recalculate and repaint the whole view according to the particular CSS of the page, which made the total process to consume a lot of time. As a betterment, React brought into the scene the virtual DOM. The **Virtual DOM** can be referred to as a copy of the actual DOM representation that is used to hold the updates made by the user and finally reflect it over to the original Browser DOM at once consuming much lesser time.

**What is ReactDOM?**

ReactDOM is a package that provides DOM specific methods that can be used at the top level of a web app to enable an efficient way of managing DOM elements of the web page. ReactDOM provides the developers with an API containing the following methods and a few more.

* render()
* findDOMNode()
* unmountComponentAtNode()
* hydrate()
* createPortal()

**Pre-requisite:** To use the ReactDOM in any React web app we must first import ReactDOM from the react-dom package by using the following code snippet:

import ReactDOM from 'react-dom'

**render() Function**

This is one of the most important methods of ReactDOM. This function is used to render a single React Component or several Components wrapped together in a Component or a div element. This function uses the efficient methods of React for updating the DOM by being able to change only a subtree, efficient diff methods, etc.

**Syntax**:

ReactDOM.render(element, container, callback)

**Parameters**: This method can take a maximum of three parameters as described below.

* **element:** This parameter expects a JSX expression or a React Element to be rendered.
* **container:** This parameter expects the container in which the element has to be rendered.
* **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.

**Return Type:** This function returns a reference to the component or null if a stateless component was rendered. 

**findDOMNode() Function**

This function is generally used to get the DOM node where a particular React component was rendered. This method is very less used like the following can be done by adding a ref attribute to each component itself.

**Syntax**:

ReactDOM.findDOMNode(component)

**Parameters**: This method takes a single parameter component that expects a React Component to be searched in the Browser DOM.

**Return Type:** This function returns the DOM node where the component was rendered on success otherwise null. 

**unmountComponentAtNode() Function**

This function is used to unmount or remove the React Component that was rendered to a particular container. As an example, you may think of a notification component, after a brief amount of time it is better to remove the component making the web page more efficient.

**Syntax**:

ReactDOM.unmountComponentAtNode(container)

**Parameters**: This method takes a single parameter container which expects the DOM container from which the React component has to be removed.

**Return Type:** This function returns true on success otherwise false. 

**hydrate() Function**

This method is equivalent to the render() method but is implemented while using server-side rendering.

**Syntax**:

ReactDOM.hydrate(element, container, callback)

**Parameters**: This method can take a maximum of three parameters as described below.

* **element:** This parameter expects a JSX expression or a React Component to be rendered.
* **container:** This parameter expects the container in which the element has to be rendered.
* **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.

**Return Type:** This function attempts to attach event listeners to the existing markup and returns a reference to the component or null if a stateless component was rendered. 

**createPortal() Function**

Usually, when an element is returned from a component’s render method, it’s mounted on the DOM as a child of the nearest parent node which in some cases may not be desired. Portals allow us to render a component into a DOM node that resides outside the current DOM hierarchy of the parent component.

**Syntax**:

ReactDOM.createPortal(child, container)

**Parameters**: This method takes two parameters as described below.

* **child:** This parameter expects a JSX expression or a React Component to be rendered.
* **container:** This parameter expects the container in which the element has to be rendered.

**Return Type:** This function returns nothing.

# JSX

## What is JSX?

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

## Coding JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement()  and/or appendChild() methods.

JSX converts HTML tags into react elements.

You are not required to use JSX, but JSX makes it easier to write React applications.

Here are two examples. The first uses JSX and the second does not:

Example 1

JSX:

const myElement = <h1>I Love JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

[Run Example »](https://www.w3schools.com/react/showreact.asp?filename=demo2_react_jsx1)

Consider this variable declaration:

const element = <h1>Hello, world!</h1>;

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

JSX produces React “elements”. We will explore rendering them to the DOM in the [next section](https://reactjs.org/docs/rendering-elements.html). Below, you can find the basics of JSX necessary to get you started.

### Why JSX?

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating technologies by putting markup and logic in separate files, React [separates concerns](https://en.wikipedia.org/wiki/Separation_of_concerns) with loosely coupled units called “components” that contain both. We will come back to components in a [further section](https://reactjs.org/docs/components-and-props.html), but if you’re not yet comfortable putting markup in JS, [this talk](https://www.youtube.com/watch?v=x7cQ3mrcKaY) might convince you otherwise.

React [doesn’t require](https://reactjs.org/docs/react-without-jsx.html) using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code. It also allows React to show more useful error and warning messages.

With that out of the way, let’s get started!

### Embedding Expressions in JSX

In the example below, we declare a variable called name and then use it inside JSX by wrapping it in curly braces:

const name = 'Josh Perez';const element = <h1>Hello, {name}</h1>;

You can put any valid [JavaScript expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions) inside the curly braces in JSX. For example, 2 + 2, user.firstName, or formatName(user) are all valid JavaScript expressions.

In the example below, we embed the result of calling a JavaScript function, formatName(user), into an <h1> element.

function formatName(user) {

return user.firstName + ' ' + user.lastName;

}

const user = {

firstName: 'Harper',

lastName: 'Perez'

};

const element = (

<h1>

Hello, {formatName(user)}! </h1>

);

### JSX is an Expression Too

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

function getGreeting(user) {

if (user) {

return <h1>Hello, {formatName(user)}!</h1>; }

return <h1>Hello, Stranger.</h1>;}

### Specifying Attributes with JSX

You may use quotes to specify string literals as attributes:

const element = <a href="https://www.reactjs.org"> link </a>;

You may also use curly braces to embed a JavaScript expression in an attribute:

const element = <img src={user.avatarUrl}></img>;

# Components and Props

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. This page provides an introduction to the idea of components. You can find a [detailed component API reference here](https://reactjs.org/docs/react-component.html).

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.

## Function and Class Components

The simplest way to define a component is to write a JavaScript function:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element. We call such components “function components” because they are literally JavaScript functions.

You can also use an [ES6 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes) to define a component:

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

The above two components are equivalent from React’s point of view.

Function and Class components both have some additional features that we will discuss in the [next sections](https://reactjs.org/docs/state-and-lifecycle.html).

## Rendering a Component

Previously, we only encountered React elements that represent DOM tags:

const element = <div />;

However, elements can also represent user-defined components:

const element = <Welcome name="Sara" />;

When React sees an element representing a user-defined component, it passes JSX attributes and children to this component as a single object. We call this object “props”.

For example, this code renders “Hello, Sara” on the page:

function Welcome(props) { return <h1>Hello, {props.name}</h1>;

}

const root = ReactDOM.createRoot(document.getElementById('root'));

const element = <Welcome name="Sara" />;root.render(element);

# React Properties

[**Last time**](https://medium.com/react-tutorials/828c397e3dc8) we looked at components, which are the building blocks of any React-powered interface. Today we will continue to explore them, with a focus on how to customise them.

There are a few ways in which React components can be customised, but we’re going pay particular attention to properties. Properties are to React components what attributes are to HTML elements. In fact, their most basic use is in the form of attributes, in JSX.

# Defining Properties

If you’re familiar with HTML (and you really should be for this tutorial to make sense) then you know that HTML elements can be customised by the attribute values contained in the opening tag.

Given a simple component:

/\*\*  
 \* @jsx React.DOM  
 \*/  
   
var InterfaceComponent = React.createClass({  
 render : function() {  
 return <div>hello world!</div>;  
 }  
});  
  
React.renderComponent(  
 <InterfaceComponent />,  
 document.body  
);

We can change the behaviour of this component by altering the contents of the **render()** method. What if we wanted the behaviour to changed, based on external information? We can use properties!

/\*\*  
 \* @jsx React.DOM  
 \*/  
   
var InterfaceComponent = React.createClass({  
 render : function() {  
 return <div>hello {this.props.name}!</div>;  
 }  
});  
  
React.renderComponent(  
 <InterfaceComponent name="chris" />,  
 document.body  
);

[**ReactJS:**](https://www.geeksforgeeks.org/react-js-introduction-working/) ReactJS is a declarative, efficient, and flexible JavaScript library for building user interfaces. It’s ‘V’ in MVC. ReactJS is an open-source, component-based front-end library responsible only for the view layer of the application. It is maintained by Facebook.

[**API:**](https://www.geeksforgeeks.org/introduction-to-apis/) API is an abbreviation for Application Programming Interface which is a collection of communication protocols and subroutines used by various programs to communicate between them. A programmer can make use of various API tools to make its program easier and simpler. Also, an API facilitates the programmers with an efficient way to develop their software programs.

Fetching data from an API in a React app

React beginners might wonder, “What exactly is an API?” To understand what an application programming interface (API) is, let’s think of an application where a section displays the daily weather forecast of the present city.

While building this type of app, we can create our backend to handle the weather data logic or we can simply make our app communicate with a third-party system that has all the weather information so we only need to render the data.

Either way, the app must communicate with the backend. This communication is possible via an API, and, in this case, a web API.

As the name implies, the API exposes an interface that our app uses to access data. With the API, we don’t need to create everything from scratch, simplifying our process. We only need to gain access to where the data is located so we can use it in our app.

## Considerations before fetching data

When we request data, we must prepare a state to store the data upon return. We can store it in a [state management tool like Redux](https://blog.logrocket.com/redux-isnt-dead/) or store it in a context object. But, to keep things simple, we will store the returned data in the React local state.

## Using the JavaScript Fetch API

The Fetch API through the fetch() method allows us to make an HTTP request to the backend. With this method, we can perform different types of operations using HTTP methods like the GET method to request data from an endpoint, POST to send data to an endpoint, and more.

Since we are fetching data, our focus is the GET method.

fetch() requires the URL of the resource we want to fetch and an optional parameter:

fetch(url, options)

We can also specify the HTTP method in the optional parameter. For the GET method, we have the following:

fetch(url, {

method: "GET" // default, so we can ignore

})

Or, we can simply ignore the optional parameter because GET is the default:

fetch(url)

As mentioned earlier, we will fetch data from a REST API. We could use any API, but here we will use a free online API [called JSONPlaceholder](https://jsonplaceholder.typicode.com/posts) to fetch a list of posts into our application; here is a [list of the resources](https://jsonplaceholder.typicode.com/) we can request

By applying what we’ve learned so far, a typical fetch() request looks like the following:

import { useState, useEffect } from "react";

export default function App() {

const [data, setData] = useState(null);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

useEffect(() => {

fetch(`https://jsonplaceholder.typicode.com/posts`)

.then((response) => console.log(response));

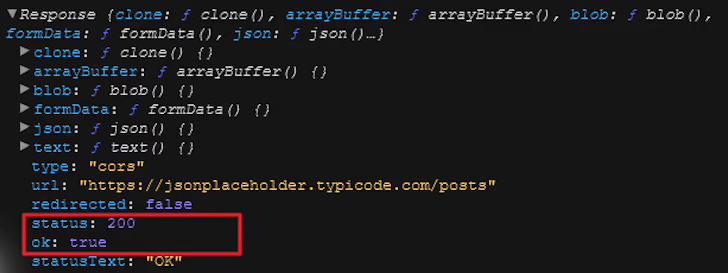
}, []);

return <div className="App">App</div>;

}

In the code, we are using the fetch() method to request post data from the resource endpoint as seen in the useEffect Hook. This operation returns a promise that could either resolve or reject.

If it resolves, we handle the response using .then(). But at this stage, the returned data is a Response object, which is not the actual format that we need, although it is useful to check for the HTTP status and to handle errors.



Take note of the Response‘s OK status; we will use it later to check for unsuccessful HTTP calls.

Next, we must resolve the Response object to JSON format using the json() method. This also returns a promise and from there, we can resolve to get the actual data that we need:

useEffect(() => {

fetch(`https://jsonplaceholder.typicode.com/posts`)

.then((response) => response.json())

.then((actualData) => console.log(actualData));

}, []);

Now, we have a list of 100 posts fetched from our API. Open the [console in this CodeSandbox](https://codesandbox.io/s/magical-rgb-qpnvp?file=/src/App.js) to see the data.

In case the promise rejects, we will handle the error using the .catch() like so:

useEffect(() => {

fetch(`https://jsonplaceholder.typicode.com/posts`)

.then((response) => response.json())

.then((actualData) => console.log(actualData))

.catch((err) => {

console.log(err.message);

});

}, []);

Note that the promise returned from the fetch() method only rejects on a network failure; it won’t reject if we hit a wrong or nonexisting endpoint like …/postssss. In this case, .catch() will not catch that error, so we must manually handle that.

Earlier we saw how the Response object returns the HTTP status. The OK status is true if we hit the correct endpoint, else it returns false. By checking for that status, we can write a custom error message for a “404 Not Found” like so:

if (!response.ok) {

throw new Error(

`This is an HTTP error: The status is ${response.status}`

);

}

And, the useEffect Hook now looks like this:

useEffect(() => {

fetch(`https://jsonplaceholder.typicode.com/posts`)

.then((response) => {

if (!response.ok) {

throw new Error(

`This is an HTTP error: The status is ${response.status}`

);

}

return response.json();

})

.then((actualData) => console.log(actualData))

.catch((err) => {

console.log(err.message);

});

}, []);

In the code block, we check if the R``esponse‘s OK status is false, meaning we have a 404 status followed by throwing our custom error message.

When we throw an error in the .then() block, .catch() detects and uses our custom message whenever we hit a “404 Not Found.”

### Rendering the posts in the frontend

Presently, we have the posts in the console. Instead, we want to render them in our app. To do that, we’ll first limit the total post number to 8 instead of the 100 posts returned for brevity.

We can do that by appending a query string parameter (?\_limit=8) to the request URL:

fetch(`https://jsonplaceholder.typicode.com/posts?\_limit=8`)

Next, we must update the state and render the UI:

// ...

export default function App() {

// ...

useEffect(() => {

fetch(`https://jsonplaceholder.typicode.com/posts?\_limit=8`)

.then((response) => {

// ...

})

.then((actualData) => {

setData(actualData);

setError(null);

})

.catch((err) => {

setError(err.message);

setData(null);

})

.finally(() => {

setLoading(false);

});

}, []);

return (

<div className="App">

<h1>API Posts</h1>

{loading && <div>A moment please...</div>}

{error && (

<div>{`There is a problem fetching the post data - ${error}`}</div>

)}

<ul>

{data &&

data.map(({ id, title }) => (

<li key={id}>

<h3>{title}</h3>

</li>

))}

</ul>

</div>

);

}

In the code, we update the state data and the error message using the setData and setError, respectively. We also added the .finally block that runs when the promise settles.

This is a good place to cancel the loading effect. Notice that we reset the error and data in the .then() and .catch(), respectively, which prevents inconsistencies for temporary server failure.

State and Lifecycle

We have seen so far what a React web-app is, the states and lifecycle of a React component. We also created a basic clock using a function to re-render the page at each second which if you think can be achieved with or without React. React doesn’t recommend using multiple renders instead it uses a stateful approach where the page is re-rendered once a state is altered.   
Our aim for this article will be to take up the code we had written in the [previous article](https://www.geeksforgeeks.org/reactjs-rendering-elements/), and create a stateful solution that will help us achieve the same result. To start let us recall what we had developed in the previous article.

import React from 'react';

import ReactDOM from 'react-dom';

function showTime()

{

const myElement = (

<div>

<h1>Welcome to GeeksforGeeks!</h1>

<h2>{new Date().toLocaleTimeString()}</h2>

</div>

);

ReactDOM.render(

myElement,

document.getElementById("root")

);

}

setInterval(showTime, 1000)

**Now what is the component in the above example ?** Actually, if you pay attention there is no component whatsoever. We are assigning a nested JSX element named “myElement” to contain the latest time and rendering the same every second, which is one of the worst ways to implement using React. Now we will start to implement it using the state and lifecycle methods which will require a classful component, let us start by creating one beforehand.

Open your react project directory and edit the**Index.js**file from src folder:

**src index.js:**

import React from 'react';

import ReactDOM from 'react-dom';

class Clock extends React.Component {

}

Now that we have defined the class to be “Clock” we must think of the process first. “Props” is the set of attributes that rarely change while “State” is the set of observable attributes that are supposed to change over time. Now if we think about our own situation, our clock has exactly two parts one is the title that says “Welcome to GeeksforGeeks!” this should be implemented using props as it will not be changing throughout the lifetime; the other part is the time itself that should be changed at each second. Let us just use the constructor and render method to first create the backbone method to show the title and time without updating it at regular intervals. 

javascript

|  |
| --- |
| import React from 'react';  import ReactDOM from 'react-dom';    class Clock extends React.Component {      constructor(props)      {          super(props);          this.state = { time : new Date() };      }        render()      {          return (              <div><h1>Welcome to { this.props.title } !</h1>          <h2>{this.state.time}</h2></div>      );    }  }    ReactDOM.render(    <Clock title="GeeksforGeeks" />,              document.getElementById('root')); |

Now that we have created our own component Clock and have rendered what we require, we only need to figure out a way of updating the time each second. Now it is clear that we have to set an interval to update the state at each second and this should be done as soon as the clock component is mounted. Thus, we have to use a lifecycle function componentDidMount() where we will set an interval to update the state, and to make the app efficient we will use componentWillUnmount() method to clear the interval. Let us see the following implementation.

Open your react project directory and edit the**Index.js**file from src folder:

**src index.js:**

import React from 'react';

import ReactDOM from 'react-dom';

class Clock extends React.Component {

constructor(props)

{

super(props);

this.state = { time : new Date() };

}

// As soon as the Clock is mounted.

// Start the interval "timer".

// Call tick() every second.

componentDidMount()

{

this.timer = setInterval(

() => this.tick(),

1000);

}

// Before unmounting the Clock,

// Clear the interval "Timer"

// This step is a memory efficient step.

componentWillUnmount()

{

clearInterval(this.timer);

}

// This function updates the state,

// invokes re-render at each second.

tick()

{

this.setState({

time : new Date()

});

}

render()

{

return (

<div><h1>Welcome to { this.props.title } !</h1>

<h2>{this.state.time.toLocaleTimeString()}</h2></div>

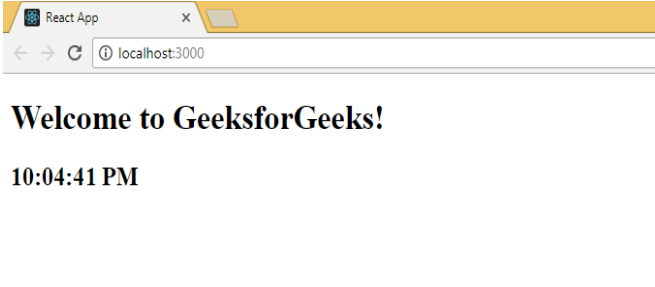
);

}

}

ReactDOM.render(

<Clock title="GeeksforGeeks" />,

document.getElementById('root')); 

JS Local storage

localStorage is a property that allows JavaScript sites and apps to save key-value pairs in a web browser with no expiration date. This means the data stored in the browser will persist even after the browser window is closed.

To use localStorage in your web applications, there are five methods to choose from:

1. setItem(): Add key and value to localStorage
2. getItem(): This is how you get items from localStorage
3. removeItem(): Remove an item by key from localStorage
4. clear(): Clear all localStorage
5. key(): Passed a number to retrieve the key of a localStorage

### setItem(): How to store values in localStorage

Just as the name implies, this method allows you to [store values in the localStorage object](https://blog.logrocket.com/storing-retrieving-javascript-objects-localstorage/).

It takes two parameters: a key and a value. The key can be referenced later to fetch the value attached to it.

window.localStorage.setItem('name', 'Obaseki Nosa');

Where name is the key and Obaseki Nosa is the value. Also note that localStorage can only store strings.

To store arrays or objects, you would have to convert them to strings.

To do this, we use the JSON.stringify() method before passing to setItem().

const person = {

name: "Obaseki Nosa",

location: "Lagos",

}

window.localStorage.setItem('user', JSON.stringify(person));

### getItem(): How to get items from localStorage

To get items from localStorage, use the getItem() method. getItem() allows you to access the data stored in the browser’s localStorage object.

getItem() accepts only one parameter, which is the key, and returns the value as a string.

To retrieve a user key:

window.localStorage.getItem('user');

This returns a string with value as:

“{“name”:”Obaseki Nosa”,”location”:”Lagos”}”

To use this value, you would have to convert it back to an object.

To do this, we make use of the JSON.parse() method, which converts a JSON string into a JavaScript object.

JSON.parse(window.localStorage.getItem('user'));

### removeItem(): How to delete localStorage sessions

To delete local storage sessions, use the removeItem() method.

When passed a key name, the removeItem() method removes that key from the storage if it exists. If there is no item associated with the given key, this method will do nothing.

window.localStorage.removeItem('name');

### clear(): How to delete all items in localStorage

Use the clear() method to delete all items in localStorage.

This method, when invoked, clears the entire storage of all records for that domain. It does not receive any parameters.

window.localStorage.clear();

### key(): How to get the name of a key in localStorage

The key() method comes in handy in situations where you need to loop through keys and allows you to pass a number or index to localStorage to retrieve the name of the key.

var KeyName = window.localStorage.key(index);

## localStorage browser support

localStorage as a type of web storage is an HTML5 specification. It is supported by major browsers including IE8. To be sure the browser supports localStorage, you can check using the following snippet:

if (typeof(Storage) !== "undefined") {

// Code for localStorage

} else {

// No web storage Support.

}

## localStorage limitations

As easy as it is to use localStorage, it is also easy to misuse it. The following are limitations, and also ways to NOT use localStorage:

* Do not store sensitive user information in localStorage
* It is not a substitute for a server based database as information is only stored on the browser
* localStorage is limited to 5MB across all major browsers
* localStorage is quite insecure as it has no form of data protection and can be accessed by any code on your web page
* localStorage is synchronous, meaning each operation called would only execute one after the other

# JavaScript Events

HTML events are **"things"** that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

## HTML Events

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

* An HTML web page has finished loading
* An HTML input field was changed
* An HTML button was clicked

Often, when events happen, you may want to do something.

JavaScript lets you execute code when events are detected.

HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

With single quotes:

<element event=**'some JavaScript'**>

With double quotes:

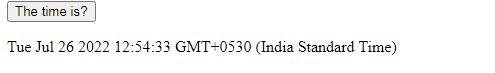
<element event=**"some JavaScript"**>

In the following example, an onclick attribute (with code), is added to a <button> element:

### Example

<button onclick="document.getElementById('demo').innerHTML = Date()">The time is?</button>

OP



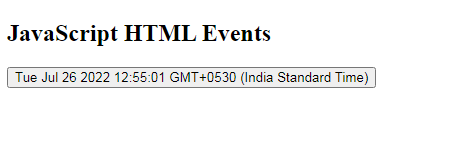
In the example above, the JavaScript code changes the content of the element with id="demo".

In the next example, the code changes the content of its own element (using **this**.innerHTML):

### Example

<button onclick="this.innerHTML = Date()">The time is?</button>

OP



JavaScript Event Handlers

Event handlers can be used to handle and verify user input, user actions, and browser actions:

* Things that should be done every time a page loads
* Things that should be done when the page is closed
* Action that should be performed when a user clicks a button
* Content that should be verified when a user inputs data
* And more ...

Many different methods can be used to let JavaScript work with events:

* HTML event attributes can execute JavaScript code directly
* HTML event attributes can call JavaScript functions
* You can assign your own event handler functions to HTML elements
* You can prevent events from being sent or being handled
* And more ...

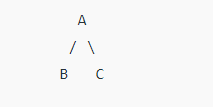
# Lifting State Up

**Lifting up the State:**As we know, every component in React has its own state. Because of this sometimes data can be redundant and inconsistent. So, by Lifting up the state we make the state of the parent component as a single source of truth and pass the data of the parent in its children.

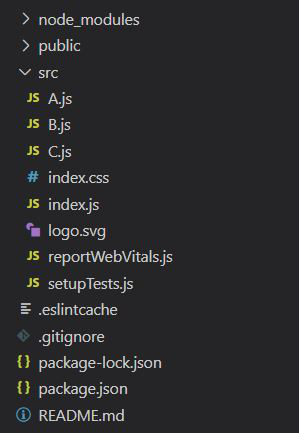
**Time to use Lift up the State:**If the data in “parent and children components” or in “cousin components” is Not in Sync.

**Example 1:**If we have 2 components in our App. **A -> B** where, A is parent of B. keeping the same data in both Component A and B might cause inconsistency of data.

**Example 2:** If we have 3 components in our App.



Where A is the parent of B and C. In this case, If there is some Data only in component B but, component C also wants that data. We know Component C cannot access the data because a component can talk only to its parent or child (Not cousins).



Approach: To solve this, we will Lift the state of component B and component C to component A. Make A.js as our Main Parent by changing the path of App in the index.js file

Before:

import App from './App';

After:

import App from './A';

Filename- A.js:

Javascript

|  |
| --- |
| import React,{ Component }  from 'react';  import B from './B'  import C from './C'    class A extends Component {      constructor(props) {      super(props);      this.handleTextChange = this.handleTextChange.bind(this);      this.state = {text: ''};    }      handleTextChange(newText) {      this.setState({text: newText});    }      render() {      return (          <React.Fragment>            <B text={this.state.text}               handleTextChange={this.handleTextChange}/>            <C text={this.state.text} />          </React.Fragment>      );    }  }    export default A; |

Filename- B.js:

Javascript

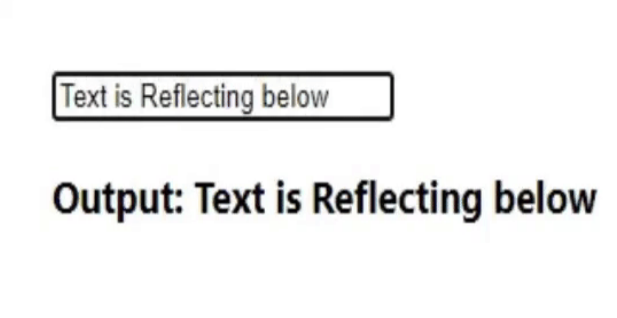
|  |
| --- |
| import React,{ Component } from 'react';    class B extends Component {    constructor(props) {      super(props);      this.handleTextChange = this.handleTextChange.bind(this);  }    handleTextChange(e){      this.props.handleTextChange(e.target.value);  }    render() {      return (          <input value={this.props.text}                 onChange={this.handleTextChange} />      );  }  }    export default B; |

Filename- C.js:

Javascript

|  |
| --- |
| import React,{ Component } from 'react';    class C extends Component {    render() {      return (          <h3>Output: {this.props.text}</h3>      );  }  }    export default C; |

Output: Now, component C can Access text in component B through component A.



Composition And Inheritance In React Js

Inheritance

Those familiar with Object Oriented Programming are well aware of Inheritance and use it on a regular basis. When a child class derives properties from it’s parent class, we call it inheritance. There are variety of use-cases where inheritance can be useful.

Example: A car is a vehicle can be modeled with inheritance.

Let’s look at a simple example of inheritance in JavaScript showing a Car is a Vehicle.

class Vehicle {

constructor (name, type) {

this.name = name;

this.type = type;

}

getName () {

return this.name;

}

getType () {

return this.type;

}

}

class Car extends Vehicle {

constructor (name) {

super(name, 'car');

}

getName () {

return 'The car's name is: ' + super.getName();

}

}

You can notice here from the code above that, the Car extends/inherits its properties from the base class, Vehicle.

Composition

Composition is also a familiar concept in Object Oriented Programming. Instead of inheriting properties from a base class, it describes a class that can reference one or more objects of another class as instances.

Example: A car has an engine can be modeled with composition.

Now that we have recapped the differences between Inheritance and Composition, let’s dive into what’s the right fit for React.

Both Inheritance and Composition, aim towards code reuse and cleaner code structure. But what does the React team recommend?

React recommends use of Composition over Inheritance, here is why. Everything in React is a component, and it follows a strong component based model. This is one of the primary reasons that composition is a better approach than inheritance for code reuse.

Take a look at the simple code snippet below to understand how composition works in React.

export default class Heading extends React.Component {

render () {

return (

<div>

<h1>{this.props.message}</h1>

</div>

)

}

}

Heading.propTypes = {

message: PropTypes.string

}

Heading.defaultProps = {

message: 'Heading One'

}

The component Heading is used to display a text message. Notice that for the Heading component the message needs to be passed in as a prop. If we fail to pass a prop, it uses the default props. The idea is that we can reuse the Heading component all across our code, and pass it a different heading message, depending on the screen that invokes it.

The code snippet below shows how Heading can be used.

// Container Component Screen One

export default class ScreenOne extends React.Component {

render () {

return (

<div>

<Heading message={'Custom Heading for Screen One'}/>

</div>

)

}

}

Notice here that ScreenOne is a container component. It uses the Heading component to display it’s heading. It passes a custom message as a prop to the Heading component as shown above.

Let’s add another container component to our code and call it ScreenTwo.

// Container Component Screen Two

export default class ScreenTwo extends React.Component {

render () {

return (

<div>

<Heading message={'Custom Heading for Screen Two'}/>

</div>

)

}

}

Notice here that ScreenTwo uses the same Heading component, but it passes a different message as prop to the component.

## What did we just demonstrate?

From the code above we demonstrated that a component can be called by another container component. And when calling a component, you can pass to it some props. We also learned that different container components can pass different props to the same component (Heading in our case).

The code above just demonstrated a simple use-case of **React Composition**.

In a [blog post](https://reactjs.org/docs/composition-vs-inheritance.html), written by Facebook they had mentioned that they had never used Inheritance in their React code across thousands of components. This shows how just using Composition can solve code reuse problems in React.

### Higher Order Components – Another Approach to Composition

A Higher Order Component (HOC) is a function that takes a component and returns a component. One use case is to inject additional props or context. This is an advanced technique that is used in React.

We can dedicate another blog post, to explore High Order Components.

### Common Components that use Composition

There are many common React Components and libraries that use Composition in React.

If you have used [react-redux](https://github.com/reduxjs/react-redux) library, it follows the same approach in the connect method. The react-redux library is used to make the redux store available to the rest of the app. If you have not used Redux or worked with it, this would be a topic we can cover in another blog post.

This composition pattern is also seen in the [react-styleable](https://github.com/pluralsight/react-styleable) component.