React-JS

**Advance React-**

 React - Styling &Amp-

**React-JS Advanced**

React is not a framework. It is just a library developed by Facebook to solve some problems that we were facing earlier. Before going into depth there are few things to understand in react, those are like Code-Splitting, Fragments, DOM, etc. Having knowledge on these topics will let you understand the core.

The Complete List of React JS Advanced Guides are listed below:

* [React Code-Splitting](https://www.geeksforgeeks.org/code-splitting-in-react/)
* [React Context](https://www.geeksforgeeks.org/context-in-react/)
* [React Fragments](https://www.geeksforgeeks.org/reactjs-fragments/)
* [React JSX In Depth](https://www.geeksforgeeks.org/react-jsx-in-depth/)
* [React refs](https://www.geeksforgeeks.org/reactjs-refs/)
* [React Creating Refs](https://www.geeksforgeeks.org/how-to-create-refs-in-reactjs/)
* [React JS Functional Components](https://www.geeksforgeeks.org/reactjs-functional-components/)
* [React JS DOM](https://www.geeksforgeeks.org/reactjs-reactdom/)
* [React JS Virtual DOM](https://www.geeksforgeeks.org/reactjs-virtual-dom/)
* [React.js Uncontrolled Vs Controlled Inputs](https://www.geeksforgeeks.org/react-js-uncontrolled-vs-controlled-inputs/)
* [Lifting State up in React JS](https://www.geeksforgeeks.org/lifting-state-up-in-reactjs/)

# Code Splitting in React

**Code-Splitting** is a feature supported by bundlers like Webpack, Rollup, and Browserify which can create multiple bundles that can be dynamically loaded at runtime.

As websites grow larger and go deeper into components, it becomes heavier. This is especially the case when libraries from third parties are included. Code Splitting is a method that helps to generate bundles that are able to run dynamically. It also helps to make the code efficient because the bundle contains all required imports and files.

**Bundling and its efficiency:** Bundling is the method of combining imported files with a single file. It is done with the help of **Webpack, Rollup, and Browserify** as they can create many bundles that can be loaded dynamically at runtime.  
With the help of code splitting, ‘lazy load’ can be implemented, which means just using the code which is currently needed.

* **The default way of importing is as follows:**

import { add } from './math';

console.log(add(x, y)); // Here x, y are two numbers

* **Using code-splitting this can be done as follows:**

import("./math").then(math => {

console.log(math.add(x, y));

});

// Here x, y are two numbers

As soon as Webpack gets this type of syntax, code-splitting is started automatically. When using the Create React App, it is already set up and can be used immediately.  
The Webpack guide on code splitting should be followed if using Webpack. The instructions can be found [here](https://webpack.js.org/guides/code-splitting/).  
When [Babel](https://www.geeksforgeeks.org/reactjs-introduction-to-babel/) is being used, it has to be made sure that Babel is not transforming the import syntax, but can parse it dynamically. This can be done using [babel-plugin-syntax-dynamic-import](https://classic.yarnpkg.com/en/package/babel-plugin-syntax-dynamic-import%20package).

### ****React.lazy and Suspense****

As both React.lazy and Suspense are not available for rendering on the server yet now, it is recommended to use [loadable-components](https://github.com/gregberge/loadable-components) for code-splitting in a server-rendered app. React.lazy is helpful for rendering dynamic import as a regular component.  
**Before:**

import Component from './Component';

**After:**

const Component = React.lazy(() => import('./Component'));

The Bundle will be loaded on its own which contains the Component when this component is rendered first.   
The Lazy component should then be rendered inside Suspense Component which helps to reflect some fallback content meanwhile the lazy component loads.

* Javascript

|  |
| --- |
| import React, { Suspense } from 'react';  const Component = React.lazy(() => import('./Component'));  function MyComponent() {  return (  <div>  <Suspense fallback={<div>Loading...</div>}>  </div>);  } |

The fallback prop can accept any element of React which will be rendered while waiting for the loading of the Component. The Suspense Component can be placed anywhere above the lazy component. Moreover, multiple lazy components can be wrapped with a single Suspense Component.

* Javascript

|  |
| --- |
| import React, { Suspense } from 'react';  const ComponentOne =  React.lazy(() => import('./ComponentOne'));  const ComponentTwo =  React.lazy(() => import('./ComponentTwo'));  function MyComponent() {  return (  <div>  <Suspense fallback={<div>Loading...</div>}>  </div>);  } |

### ****Error Boundaries****

[Error Boundaries](https://www.geeksforgeeks.org/react-js-error-boundaries/) are React components that help when some modules fail to load due to any issue, an error will be triggered. These errors can be handled properly and provide a good experience to the user by the use of a suitable error page.

* Javascript

|  |
| --- |
| import React, { Suspense } from 'react';  import ErrorBoundary from './ErrorBoundary';  const ComponentOne = React.lazy(() =>  import('./ComponentOne'));  const ComponentTwo = React.lazy(() =>  import('./ComponentTwo'));  const MyComponent = () => (  <div>  <Suspense fallback={<div>Loading...</div>}>  </div>  ); |

### ****Route-Based Code Splitting****

It can be difficult to implement code-splitting in code, the bundles can be split evenly, which will improve the experience for the user.  
Here you can see the example code for this.

* Javascript

|  |
| --- |
| import React,{Suspense, lazy} from 'react';  import {Route, Switch, BrowserRouter } from 'react-router-dom';  const HomePage = lazy(() =>  import('./routes/HomePage'));  const AboutUs = lazy(() =>  import('./routes/AboutUs'));  const App = () =>  (<Suspense fallback={<div>Loading...</div>}>); |

### Named Exports

React.lazy currently supports only default exports. An intermediate module that re-exports as default has to be created if one wants to import a module that uses named exports. This ensures the working of tree shaking and prevents the pulling in of unused components.

* Javascript

|  |
| --- |
| import {React, lazy} from 'react';  // Components.js  export const Component = /\* ... \*/;  export const MyUnusedComponent = /\* ... \*/;  // Component.js  export { Component as default } from "./Components.js";  // MyApp.js  const Component = lazy(() => import("./Component.js")); |

# Context in React

In this article, you will be introduced to React Context, one of the latest features in React Applications.

**Prerequisites:**

* [ReactJS-Basics](https://www.geeksforgeeks.org/reactjs-basics-concepts-complete-reference/),
* [Typescript](https://www.geeksforgeeks.org/introduction-to-typescript/)(with Interfaces),
* [ES6](https://www.geeksforgeeks.org/introduction-to-es6/) notation

**What is React Context?**  
React Context is a method to pass props from parent to child component(s), by storing the props in a store(similar in Redux) and using these props from the store by child component(s) without actually passing them manually at each level of the component tree.

**Why React Context? We have Redux!!**  
Using [Redux](https://www.geeksforgeeks.org/introduction-to-react-redux/) to interact with states from parent to child components is not only quite difficult to understand but also gives you a more complex code. Through the usage of Context, the understanding of concept and code is far easier than that of Redux.

**When to use React Context?**  
Anytime you want! There is no iron-clad rule like when to use Context in your application. Whenever you want a store to keep your states or variables in and use them elsewhere in your program, use Context. Generally, when we have two or more levels(height) in our component tree, it is viable to use a store instead of passing props and then lifting the state as this will create confusion and unnecessary lengthy code.   
**Example:**  
If we have three components in our app, A->B->C where A is the parent of B and B is the parent of C. To change a state from C and pass it to A, keep the state of A in a store, then extract the state from the store and use it in C. This completely eliminates the necessity of the state to pass through B. So the flow is like A->C.

**Usage**  
Now let’s understand with a simple example.   
first, let’s define an interface(or class) consisting of name and marks of a student:

interface MarksContext{

name: string;

marks: number;

}

**Now define this interface in a Context.**

const contextmarks = React.createContext(null);

// Context defined with properties of MarksContext interface initialized with null

**Create Provider and Consumer**

const MarksContextProvider = contextmarks.Provider;

// This is the store in which states will be kept in and passed as props.

const MarksContextConsumer = contextmarks.Consumer;

// This is the store from which states stored in the Provider

// Context will be extracted by child component and used according to the user.

Now, let our component tree structure be like A->B->C(from above). We will store some value of name and marks in a dictionary, pass the info from A to display it in C without meddling with B. A is the root App.

Complete code:

**Note:**keep the two files in the same directory.

**1. MarksContext.tsx(typescript)**

* Javascript

|  |
| --- |
| // MarksContext.tsx(typescript) File  import \* as React from "react";    export interface MarksContext {      name: string;      marks: number;  }  const contextmarks = (React.createContext < MarksContext) | (null > null);  export const MarksContextProvider = contextmarks.Provider;  export const MarksContextConsumer = contextmarks.Consumer; |

**2. App.tsx(typescript)**

* Javascript

|  |
| --- |
| import \* as React from "react";  import { render } from "react-dom";  import { MarksContext, MarksContextProvider } from "./MarksContext";  import { MarksContextConsumer } from "./MarksContext";    const sample: MarksContext = {      name: "X",      marks: 20,  };    export const A = () => (      <MarksContextProvider value={sample}>          <B />      </MarksContextProvider>  );    const B = () => (      <div>          <h2>Student Info</h2>          <C />      </div>  );    const C = () => (      <MarksContextConsumer>          {(appContext) =>              appContext && (                  <div>                      Name: {appContext.name} <br />                      Marks: {appContext.marks} <br />                  </div>              )          }      </MarksContextConsumer>  );  render(<A />, document.getElementById("root")); |

**Output:**You can see the output of this on the browser it will be the same as shown below.



**Working:**   
The information is first declared in an interface *MarksContext*, then the interface information is defined in *contextmarks*. Provider(*MarksContextProvider*) and consumer(*MarksContextConsumer*) are defined in this context.   
*MarksContextProvider* is put in *A*, the root of our app with *value* initialized as given in *sample*.Later the *MarksContextConsumer* is put in C, from which value of interface is extracted through *appContext* which serves as an instance of the *MarksContext* interface. Lastly, the value is displayed in C

# React JSX in Depth

As we have seen in this article [React.js (Introduction and Working)](https://www.geeksforgeeks.org/react-js-introduction-working/), how React works, Breaking down our application into smaller reusable parts which we call ***components***. These components are tree-like HTML or Component structure. In that article, we have seen how to create elements using ***createElement***in react. But if we’ve to work with React’s createElement method just to create elements then it would be fine for demo purposes or for a small app but not for a large application. You just should not use it.

React.createElement is fine to create elements for a small app or for demo purposes but not for a large salable app because it would be very hard to maintain or debug. we’ve to call React.createElement method every time for creation of React element even if it just a span tag with no attributes.

You can build whole application using React.createElement method(which I don’t recommend).

If we shouldn’t use **createElement**use JSX. It is JavaScript and XML. It is a JavaScript extension that allows us to define React elements using a tag-based style syntax within our JS code. How cool is that?

# ReactJS Refs

* **ReactJS Refs**are used to access and modify the **DOM elements** in the React Application. It creates a reference to the elements and uses it to modify them.

Table of Content

* [What are React refs ?](https://www.geeksforgeeks.org/reactjs-refs/#what-are-react-refs-)
* [Creating refs in React](https://www.geeksforgeeks.org/reactjs-refs/#creating-refs-in-react)
* [Accessing Refs in React](https://www.geeksforgeeks.org/reactjs-refs/#accessing-refs-in-react)
* [When to use refs](https://www.geeksforgeeks.org/reactjs-refs/#when-to-use-refs)
* [When not to use refs](https://www.geeksforgeeks.org/reactjs-refs/#when-not-to-use-refs)

## ****What is Refs in React?****

**Refs**are a function provided by [**React**](https://www.geeksforgeeks.org/react-tutorial/)to access the [**DOM element**](https://www.geeksforgeeks.org/dom-document-object-model/) and the React elements created in [**components**](https://www.geeksforgeeks.org/reactjs-components/). They are used in cases where we want to change the value of a child component, without making use of [**props**](https://www.geeksforgeeks.org/reactjs-props-complete-reference/)and [**state**](https://www.geeksforgeeks.org/reactjs-state/).

They allow us to interact with these elements outside the typical rendering workflow of React.

# React JS React-DOM

React JS ReactDOM or react-dom is the most widely used package of React. React provides the developers with a package **react-dom** to access and modify the DOM. Let’s see in brief what is the need to have the package.

Table of Content

* [What is ReactDOM ?](https://www.geeksforgeeks.org/reactjs-reactdom/#what-is-reactdom-)
* [How to use ReactDOM ?](https://www.geeksforgeeks.org/reactjs-reactdom/#how-to-use-reactdom-)
* [Why ReactDOM is used ?](https://www.geeksforgeeks.org/reactjs-reactdom/#why-reactdom-is-used-)
* [Important functions provided by ReactDOM](https://www.geeksforgeeks.org/reactjs-reactdom/#important-functions-provided-by-reactdom)
* [Key features of ReactDOM](https://www.geeksforgeeks.org/reactjs-reactdom/#key-features-of-reactdom-)

## ****What is React-DOM ?****

ReactDOM is a package in React 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 various methods to manipulate DOM.

## How to use React-DOM ?

To use the React-DOM in any React web app we must first install the react-dom package in our project. To install the react-dom package use the following command.

# React-JS Virtual DOM

React JS Virtual DOM is an in-memory representation of the DOM. DOM refers to the Document Object Model that represents the content of XML or HTML documents as a tree structure so that the programs can be read, accessed and changed in the document structure, style, and content.

Table of Content

* [Prerequisites](https://www.geeksforgeeks.org/reactjs-virtual-dom/#prerequisites)
* [What is DOM ?](https://www.geeksforgeeks.org/reactjs-virtual-dom/#what-is-dom-)
* [Disadvantages of real DOM](https://www.geeksforgeeks.org/reactjs-virtual-dom/#disadvantages-of-real-dom-)
* [Virtual DOM](https://www.geeksforgeeks.org/reactjs-virtual-dom/#virtual-dom)
* [How does virtual DOM actually make things faster?](https://www.geeksforgeeks.org/reactjs-virtual-dom/#how-does-virtual-dom-actually-make-things-faster)
* [How virtual DOM Helps React?](https://www.geeksforgeeks.org/reactjs-virtual-dom/#how-virtual-dom-helps-react)
* [Virtual DOM Key Concepts](https://www.geeksforgeeks.org/reactjs-virtual-dom/#virtual-dom-key-concepts-)
* [Differences between Virtual DOM and Real DOM](https://www.geeksforgeeks.org/reactjs-virtual-dom/#differences-between-virtual-dom-and-real-dom)

## ****What is DOM ?****

DOM stands for ‘Document Object Model’. In simple terms, it is a structured representation of the HTML elements that are present in a webpage or web app. DOM represents the entire UI of your application. The DOM is represented as a tree data structure. It contains a node for each UI element present in the web document. It is very useful as it allows web developers to modify content through JavaScript, also it being in structured format helps a lot as we can choose specific targets and all the code becomes much easier to work with.

## Disadvantages of real DOM :

Every time the DOM gets updated, the updated element and its children have to be rendered again to update the UI of our page. For this, each time there is a component update, the DOM needs to be updated and the UI components have to be re-rendered

React - Styling &Amp-

There are many ways to style React with CSS, this tutorial will take a closer look at three common ways:

* Inline styling
* CSS stylesheets
* CSS Modules

**Inline Styling**

To style an element with the inline style attribute, the value must be a JavaScript object:

### Example:[Get your own React.js Server](https://www.w3schools.com/react/react_server.asp" \o "W3Schools Spaces" \t "_blank)

Insert an object with the styling information:

const Header = () => {

return (

<>

<h1 style={{color: "red"}}>Hello Style!</h1>

<p>Add a little style!</p>

</>

);

}

**Note:** In JSX, JavaScript expressions are written inside curly braces, and since JavaScript objects also use curly braces, the styling in the example above is written inside two sets of curly braces {{}}.

### camelCased Property Names

Since the inline CSS is written in a JavaScript object, properties with hyphen separators, like background-color, must be written with camel case syntax:

### Example:

Use backgroundColor instead of background-color

const Header = () => {

return (

<>

<h1 style={{backgroundColor: "lightblue"}}>Hello Style!</h1>

<p>Add a little style!</p>

</>

);

}

### JavaScript Object

You can also create an object with styling information, and refer to it in the style attribute:

### Example:

Create a style object named myStyle:

const Header = () => {

const myStyle = {

color: "white",

backgroundColor: "DodgerBlue",

padding: "10px",

fontFamily: "Sans-Serif"

};

return (

<>

<h1 style={myStyle}>Hello Style!</h1>

<p>Add a little style!</p>

</>

);

}

## CSS Stylesheet

You can write your CSS styling in a separate file, just save the file with the .css file extension, and import it in your application.

### App.css:

Create a new file called "App.css" and insert some CSS code in it:

body {

background-color: #282c34;

color: white;

padding: 40px;

font-family: Sans-Serif;

text-align: center;

}

**Note:** You can call the file whatever you like, just remember the correct file extension.

Import the stylesheet in your application:

### index.js:

import React from 'react';

import ReactDOM from 'react-dom/client';

import './App.css';

const Header = () => {

return (

<>

<h1>Hello Style!</h1>

<p>Add a little style!.</p>

</>

);

}

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

root.render(<Header />);

## CSS Modules

Another way of adding styles to your application is to use CSS Modules.

CSS Modules are convenient for components that are placed in separate files.

The CSS inside a module is available only for the component that imported it, and you do not have to worry about name conflicts.

Create the CSS module with the .module.css extension, example: my-style.module.css.

.bigblue {

color: DodgerBlue;

padding: 40px;

font-family: Sans-Serif;

text-align: center;

}

Import the stylesheet in your component:

### Car.js:

import styles from './my-style.module.css';

const Car = () => {

return <h1 className={styles.bigblue}>Hello Car!</h1>;

}

export default Car;

Import the component in your application:

### index.js:

import ReactDOM from 'react-dom/client';

import Car from './Car.js';

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

root.render(<Car />);

# Introduction

AWS Amplify is a complete solution that lets frontend web and mobile developers easily build, connect, and host fullstack applications on AWS, with the flexibility to leverage the breadth of AWS services as your use cases evolve. Amplify provides the following products to build fullstack iOS, Android, Flutter, Web, and React Native apps. These products are all designed to work independently:

* **Amplify**[**CLI**](https://docs.amplify.aws/gen1/react/tools/cli/) - Configure all the services needed to power your backend through a simple command line interface.
* **Amplify**[**Libraries**](https://docs.amplify.aws/gen1/react/tools/libraries/) - Client libraries to build common use cases such as Auth, data, and File Storage by connecting your frontend app to your backend resources.
* **Amplify**[**Studio**](https://docs.amplify.aws/gen1/react/tools/console/) - Build your fullstack web and mobile app using a simple and intuitive visual development environment.
* **Amplify**[**UI Components**](https://ui.docs.amplify.aws/) - UI libraries to build your frontend app built for React, React Native, Angular, Vue and Flutter.
* **Amplify**[**Hosting**](https://aws.amazon.com/amplify/hosting/) is an AWS service that provides a git-based workflow for continuous deployment & hosting of fullstack web apps.

## [What you'll build](https://docs.amplify.aws/gen1/react/start/getting-started/introduction/#what-youll-build)

This tutorial guides you through setting up a backend and integrating that backend with your React app. You will create a “Todo app” with a GraphQL API and to store and retrieve items in a cloud database. In addition, we’ll demonstrate how to authenticate users, communicate with your API, and set up continuous deployment and hosting.

[GraphQL](http://graphql.org/) is a data language that was developed to enable apps to fetch data from APIs. It has a declarative, self-documenting style. In a GraphQL operation, the client specifies how to structure the data when it is returned by the server. This makes it possible for the client to query only for the data it needs, in the format that it needs it in.

Congratulations! Your app is built, with a realtime backend.

What next? Here are some things to add to your app:

* [Authentication](https://docs.amplify.aws/gen1/react/build-a-backend/auth/set-up-auth/)
* [User File Storage](https://docs.amplify.aws/gen1/react/build-a-backend/storage/set-up-storage/)
* [Serverless APIs](https://docs.amplify.aws/gen1/react/build-a-backend/graphqlapi/set-up-graphql-api/)
* [Analytics](https://docs.amplify.aws/gen1/react/build-a-backend/more-features/analytics/set-up-analytics/)
* [AI/ML](https://docs.amplify.aws/gen1/react/build-a-backend/more-features/predictions/set-up-predictions/)
* [PubSub](https://docs.amplify.aws/gen1/react/build-a-backend/more-features/pubsub/set-up-pubsub/)