

# React

## What is React? Why React is known as 'React'?

- **React** is a popular **JavaScript library** for building **user interfaces**, particularly for **single-page applications**. It was developed by **Facebook** and is now maintained by **Facebook** and a community of individual developers and companies.
- The name '**React**' was chosen because the **library** was designed to **allow developers to react to changes in state and data within an application**, and to update the **user interface** in a **declarative** and **efficient manner**.

## What are Library and Frameworks?

### Library: -


- **Library** is a **collections of prewritten code snippets** that can be **used** and **reused** to **perform** common tasks without having to **write code** from **scratch**.

### Framework: -

- A **framework** is a **pre-built structure** that provides a **foundation** and **guidelines** for developing **software applications**. It offers a **set of tools, libraries**, and **conventions** that define how an **application** should be **organized** and **built**, often controlling the overall program flow.

## Differences between Library and framework: -

**Library** and **frameworks** are both **pre-written code that developers use to save time and effort when building software applications**. However, there are some key differences between the two:

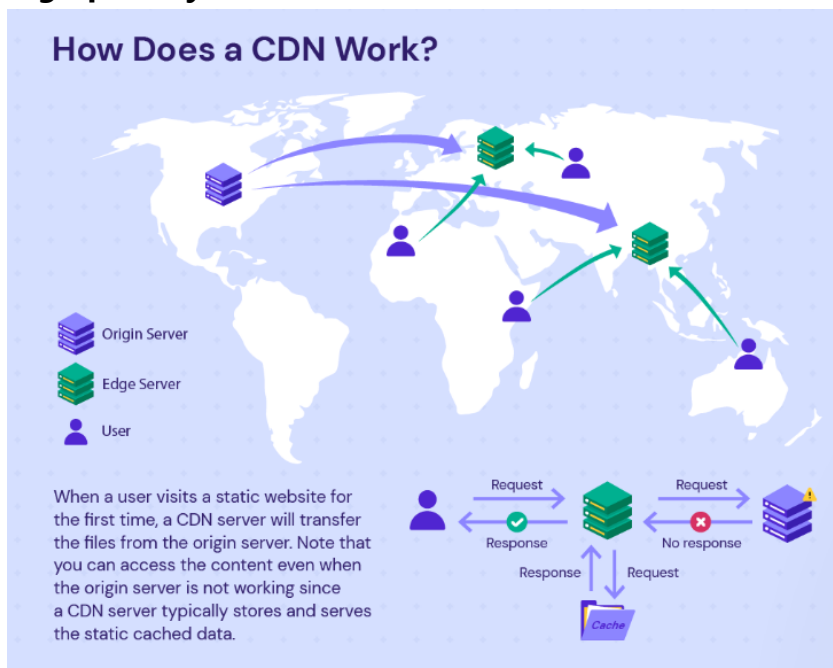
Feature	Library	Framework
Definition	A library is a collection of pre-written code that provides reusable functionalities. You decide when and how to use its components in your program.	A framework is a pre-established structure that dictates the overall architecture and flow of your application. It calls your code, and you fill in specific functionalities.
Control Flow	You maintain control over the flow of your program using the library's functions.	The framework dictates the flow of control; you provide implementations for the framework to call.
Inversion of Control (IoC)	You call functions from the library when needed, maintaining a more straightforward control flow.	The framework calls your code, inverting control to adhere to its structure and conventions.
Flexibility	More flexible; you can pick and choose specific components to integrate into your application.	More opinionated; follows a specific structure and may limit flexibility but provides consistency.
Size and Scope	Smaller in scope, focused on specific tasks or functionalities.	Comprehensive, providing a broader set of tools and conventions for building entire applications.
Examples	jQuery, NumPy, Requests, Ret  React (considered a library).	Django, Flask, Ruby on Rails, Spring.

## What Is a CDN?

A **content delivery network (CDN)** is a **group of servers** spread across different **geographical locations worldwide** to enable the quick delivery of a website's content. It is also known as a **content distribution network**.

## How Does a CDN Work?

To understand how a **CDN** works, we need to know about **origin** and **edge web servers**. An **origin server** hosts the **original website files**. An **edge server caches** (A **cache** is a **temporary storage area** that keeps **copies of data** for **faster future access**.) copies of this content retrieved from the origin server. These edge servers are located in **data centers worldwide**, forming a **geographically distributed network**.



To help us understand the advantages of using a **CDN**, let's compare the **web content delivery** process between two websites: one **without a CDN** and **one using one**.

- When a user accesses a website without a **CDN**, the browser connects to the **origin server** and **requests site content**. In addition to delivering the requested data, the **origin server** responds to each subsequent user **request**. Static files are cached locally on the user's computer.
- The entire process consists of the user sending requests to the **origin server** and the **origin server responding** by delivering **website content**. This procedure remains the same regardless of the user's **geographical location**.
- A **problem arises** when the **end-user** is far away from the **origin server**. The farther the distance between the **two**, the longer it will take for the **page to load**.
- In addition, as the **origin server handles all user requests**, **web performance** may suffer when the workload increases due to **traffic spikes**.
- In contrast, when a user accesses a **website with a CDN**, the **browser connects** to one of the **edge servers** to **request site content**. The nearest **server** is typically chosen to

**minimize time delay.** Using multiple **caching servers** distributes **traffic** and prevents **server overload**.

- The **edge server** will then forward the **request** to the **origin server**. After getting the data from the **origin server**, the **edge server delivers** it to the **end-user** and **caches** the **files locally**.
- This means the **edge server** will **serve all subsequent requests** using the **cached** files without having to **fetch data** from the **origin server again**. The Internet content stored on the **network edge** can also be delivered to the **end-user** even if the **origin server** becomes inaccessible.
- As such, using a **CDN** may lead to **better web performance** and **user experience**.

### Understanding CDN Services by Example:

- Imagine we operate an **online store** based in the **United States**. our **eCommerce website** is **hosted** on an **origin server** located in **New York**, where all our website files are stored.
- Without a **CDN**, a customer accessing our **website** from **Tokyo** will most likely face a **slow-loading site**. This is because there is a large distance between the **site visitor's geographical location** and our site's **origin server**.
- Using a **CDN** will speed up the **page load times**. With a **CDN**, when the **Tokyo-based user** requests content from our **website**, the **browser** will retrieve data from one of the **CDN's edge servers** or points of presence spread worldwide, typically the one closest to the **user's location**.
- Connecting to a nearby **CDN server decreases** the **distance** between the **user** and the **server**, improving the **content delivery speed**.
- Therefore, our site **visitors** will get to enjoy **fast site speed** regardless of their geographical location.

### What are the benefits of using a CDN?

Although the benefits of using a **CDN** vary **depending** on the **size** and **needs** of an **Internet property**, the primary benefits for most users can be broken down into 4 different components:

- **Improving website load times** - By **distributing content** closer to **website visitors** by using a nearby **CDN server** (among other optimizations), visitors experience **faster page loading times**. As visitors are more inclined to click away from a **slow-loading site**, a **CDN** can reduce **bounce rates** and increase the amount of time that people spend on the site. In other words, a **faster website** means **more visitors** will stay and stick around longer.
- **Reducing bandwidth costs** - Bandwidth consumption costs for **website hosting** is a primary expense for **websites**. Through **caching** and **other optimizations**, **CDNs** are able to **reduce** the amount of **data** an **origin server** must provide, thus reducing hosting costs for website owners.
- **Increasing content availability and redundancy** - large amounts of **traffic** or **hardware failures** can interrupt normal **website function**. Thanks to its distributed nature, a **CDN** can handle more **traffic** and withstand hardware failure better than many **origin servers**.

- **Improving website security** - A **CDN** may improve security by providing **DDoS(Distributed Denial of Service) mitigation**, improvements to security certificates, and other optimizations.

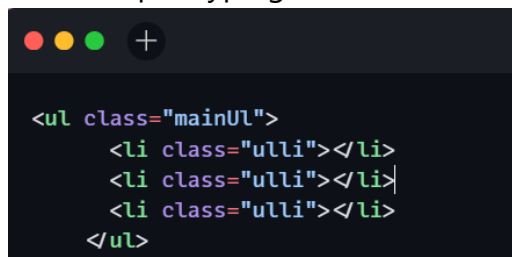
## The main reasons to avoid using CDN links in React projects are: -

1. **Lack of Dependency Management:** CDNs don't manage versions or track dependencies like npm, leading to version mismatches and manual updates.
2. **No Build Optimizations:** CDNs don't allow for features like **tree shaking**, **minification**, or **code splitting**, resulting in less efficient, larger bundles.
3. **Limited Development Tools:** CDNs don't integrate with tools like **Webpack** or **Vite** for **hot reloading**, **debugging**, or **build customization**.
4. **Security Risks:** Relying on third-party CDNs exposes us to potential security vulnerabilities.
5. **No Offline Development:** CDNs require an internet connection, limiting development in offline environments. CDN links often load the entire library, even if we only need a small portion. This leads to unnecessary code bloat and larger initial load times
6. **No Development/Production Differentiation:** CDN links don't automatically provide optimized production builds or debugging features for development.

## What is Emmet?

**Emmet** is a powerful **toolkit** for **web developers** that significantly **speeds** up **HTML** and **CSS** **coding**. It allows **developers** to **write concise, abbreviated syntax** that **expands** into **full-fledged code snippets**.

For example, typing "**ul.mainUI>li.ulli\*3**" and expanding it with Emmet would instantly generate:



```
<ul class="mainUI">
  <li class="ulli"></li>
  <li class="ulli"></li>
  <li class="ulli"></li>
</ul>
```

For one more example, typing "**!**" and expanding it with Emmet would instantly generate:



```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body></body>
</html>
```

## What is difference between React and ReactDOM?

**React** and **ReactDOM** are both **libraries** that are used in the **development** of **user interfaces**, but they serve different **purposes**. Here's a breakdown of the differences between the two:

### React:

- **React** is the **core library** for building **user interfaces**.
- It contains the functionality for **defining** and **creating** React components.
- It provides features like **state management**, **props**, and the **component** lifecycle.
- **React** can be used for both **web** and **mobile development** (React Native).

### ReactDOM:

- **ReactDOM** is a complementary library specifically for **web applications**.
- It provides **DOM-specific methods** that enable **React** to interact with the **browser's DOM**.
- It's responsible for **rendering** React **components** to the **DOM**.
- It's only used in **web browser** environments.

### Key differences:

#### 1. Purpose:

- **React:** Component creation and management
- **ReactDOM:** DOM manipulation and rendering

#### 2. Usage:

- **React:** import **React** from 'react';
- **ReactDOM:** import **ReactDOM** from 'react-dom';

#### 3. Common methods:

- **React:** createElement(), Component, useState(), useEffect()
- **ReactDOM:** render(), createPortal()

#### 4. Platform:

- **React:** Cross-platform (web, mobile)
- **ReactDOM:** Web-only

Example usage:

```
const heading = React.createElement(
  "h1",
  { id: "Heading" },
  "Hello from React"
);

const root = ReactDOM.createRoot(document.querySelector(".root"));

root.render(heading);
```

In this example, we use **React** to **define** the **React Element** and **ReactDOM** to **render** it to the **DOM**.

## What is React.createElement?

React.createElement is a **function** provided by React that is used to create **React elements**. It accepts several parameters and returns a new element of the specified type. React elements are the **building blocks** of React components. In its simplest form, it is used as follows:

```
React.createElement(type, [props], [...children]);
```

Under the hood, React.createElement first creates a **new object** with the following properties:

- **Type:** This can be a string representing a DOM tag (like "div", "span", etc.) or a React component (**function** or **class**).
- **Props:** An object that contains the **properties** of the element. This can be anything from **className**, **style**, or any **custom props**. These properties are passed to the React Element when it is rendered.
- **children:** The content inside the element. It can be **strings**, **numbers**, other **React elements**, or an **array** of these. Multiple child elements can be passed as separate arguments.

Here is an example of how React.createElement is used to create a React element:

```
const element = React.createElement("h1", { className: "h1" }, "Inside H1");
```

- The first argument to **React.createElement** is the **type of the element**. In this case, the type is **"h1"**.
- The second argument is an **object** that contains the properties of the element. In this case, the only property is **className**. The value of className is **"h1"**.

- The third argument is **an array of child elements**. In this case, there is only one child element, which is the string **"Inside H1"**.
- The result of `React.createElement` is a **new React element instance**. The **React element instance** can then be used to **render** the element in the DOM.

## What is React Element: -

A React element is a **plain JavaScript object** that **describes what to render on the screen**. It is the building block of React components.

Under the hood, a React element is an object with the following properties:

- **Type**: The type of the element, which can be a **string** (like **'div'**, **'span'**) for **HTML elements** or a **function/class** for custom **React components**.
- **Props**: An object that contains the **properties of the element**. These properties are passed to the **React component** when it is rendered.
- **Children**: An array of child elements. These child elements are rendered **inside the parent element**.

For example, the following **React element** describes a **div** element and **h1** and **h2** element with the text **"Inside h1"** and **"Inside h2"**:

```
const newElement = React.createElement("div", { className: "rootDiv" }, [  
  React.createElement("h1", { id: "h1" }, "Inside h1"),  
  React.createElement("h2", { id: "h2" }, "Inside h2"),  
]);
```

- The **type property** of the element is **div**. This means that the element will be rendered as a **div element**.
- The **props property** of the element contains the **properties of the element**. In this case, the only property is **className**. The value of **className** is **"rootDiv"**.
- The **children property** of the element contains an **array** of child elements. In this case, there are two child elements. The first child element is a React element that describes a **h1** element with the text **"Inside h1"**. The second child element is a React element that describes a **h2** element with the text **" Inside h2"**.
- When a React element is rendered, the React framework will create a DOM node for the element and then render the children of the element into the **DOM node**.

```
const reactElement = React.createElement("div",{className:"mainDiv"},"inside Div");  
  
console.log(reactElement);
```

The above code creates an object that looks like this:



## Why Use React.createElement?

While **React.createElement** may seem verbose, it's extremely powerful. It's the base of **JSX syntax** which is more common and easier to write. The following JSX code:

```
const heading = (
  <div className="App">
    <h1>Hello, world</h1>
  </div>
);
```

is syntactic sugar for:

```
React.createElement(
  "div",
  { className: "App" },
  React.createElement("h1", null, "Hello, World!")
);
```

Understanding the usage of **React.createElement** can give a better understanding of what happens under the hood when we write **JSX**.

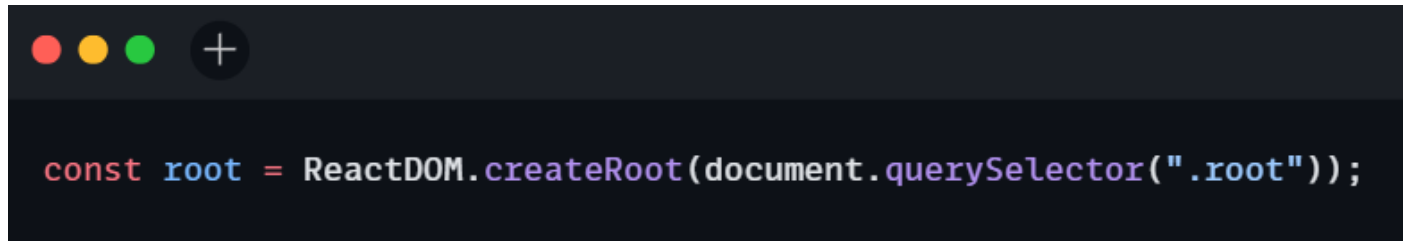


## ReactDOM.createRoot: -

ReactDOM.createRoot is a **function** that **creates a new root DOM node** for rendering React Elements.

Under the hood, ReactDOM.createRoot first creates a new object with the following properties:

- **Container:** The DOM element that the **React elements** should be rendered in.
- **Updater:** A function that is used to **update the DOM** when the React elements are



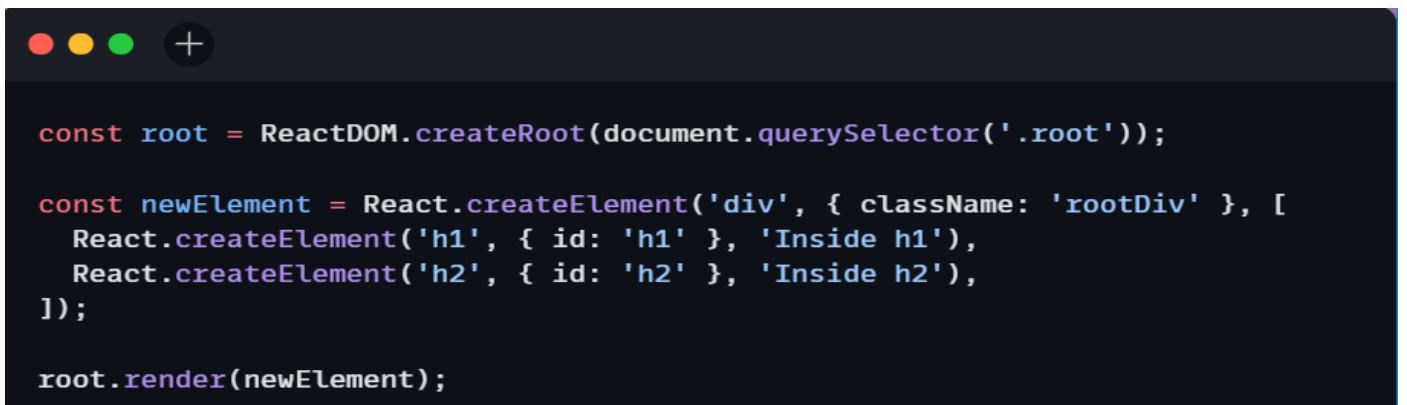
```
const root = ReactDOM.createRoot(document.querySelector(".root"));
```

updated.

Here is an example of how ReactDOM.createRoot is used to create a new root DOM node: -

- The first argument to ReactDOM.createRoot is the DOM element that the **React elements should be rendered in**. In this case, the DOM element is the div element with the id **root**.
- The result of ReactDOM.createRoot is a new ReactDOM.Root instance. The ReactDOM.Root instance can then be used to **render React elements in the DOM**.

Here is an example of how the ReactDOM.Root instance can be used to render React elements in the DOM: -



```
const root = ReactDOM.createRoot(document.querySelector('.root'));

const newElement = React.createElement('div', { className: 'rootDiv' }, [
  React.createElement('h1', { id: 'h1' }, 'Inside h1'),
  React.createElement('h2', { id: 'h2' }, 'Inside h2'),
]);

root.render(newElement);
```

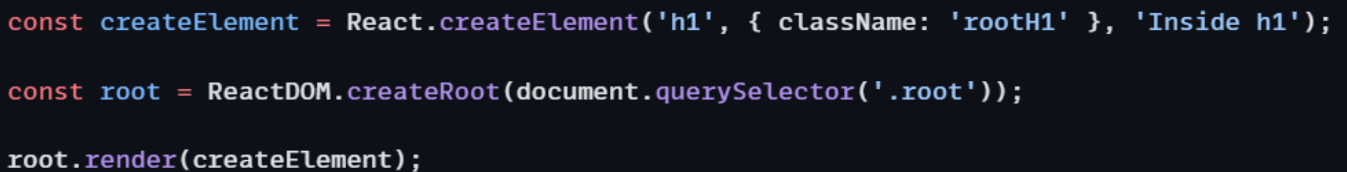
In this example, the React element instance is passed to the **ReactDOM.Root.render** function. The ReactDOM.Root.render function takes one argument: the React element instance that should be rendered in the root DOM node.

The ReactDOM.Root.render function will then **render the React element instance in the root DOM node**.

## render() Function: -

- The render is a **function** that is used to render a React element or a tree of elements into the root created by ReactDOM.createRoot.
- When we use ReactDOM.render(), it **replaces whatever is inside the root container**; it does not **append content** but rather **replaces the existing content**.
- The render method takes no arguments and returns a **React element**. The **React element** that is returned by the **render method** is **what is rendered on the screen**.

Example:



```
const createElement = React.createElement('h1', { className: 'rootH1' }, 'Inside h1');  
  
const root = ReactDOM.createRoot(document.querySelector('.root'));  
  
root.render(createElement);
```

## Here are some additional things to keep in mind about the render method:

- The render method must return a **React element**.
- The render method can be called **multiple times**, but the React element that is returned will always be the same.
- The render method can be nested. This means that we can call the **render method** from within **another render method**.
- The render method can be used **to create dynamic content**. This means that we can change the **content of the React element that is rendered based on the current state of the component**.

## Workflow of render() method:-

- The render() method is called by the **ReactDOM** library.
- The render() method returns a **React element**.
- The React framework creates a **DOM node for the React element**.
- The React Library renders the **children of the React element into the DOM node**.

## The Role of Babel: -

Babel is a **JavaScript compiler** that takes our modern JavaScript code and returns **browser compatible JavaScript**. It translates JSX into **React.createElement**. This is one of the reasons why setting up a React environment requires a bit of configuration.