

#20daysofReact

By

@MAYBESHALINII

Topics Covered:

- ☐ Intro to React
- ☐ Components and Props
- ☐ States and Prop drilling
- ☐ Conditional Rendering and Component Lifecycle
- ☐ Hooks
- ☐ Nested Routes
- ☐ Context API
- ☐ Redux
- ☐ Fetch API
- ☐ Error Handling
- ☐ Debouncing
- ☐ Higher Order Components
- ☐ Structure Your Frontend Code
- ☐ Concurrent React

- ☐ Server and Client Side Rendering
- ☐ Interview questions

The purpose of adding check boxes here is to help you keep track of which topics you have covered and which are left.

Any doubts on any topics are welcome. You can dm me at @maybeshalinii or you can mail me at [@tewarishalini02@gmail.com](mailto:tewarishalini02@gmail.com)

This doc might not be “eye pleasing or aesthetically beautiful” but it will surely help you gain knowledge.

Introduction to React

What is React?

React is a popular JavaScript library for building user interfaces. It was developed by Jordan Walke, a software engineer at Facebook, and is widely used for creating dynamic and interactive web applications. React makes it easier to manage the state of your application, handle user interactions, and efficiently update the UI when data changes. It follows a component-based architecture, allowing you to build complex UIs by composing reusable components.

Why was react needed?

React was created because building complex user interfaces using just plain JavaScript can become messy and hard to manage as projects grow. React makes this easier by providing a structured way to create user interfaces.

Imagine you're building a house with lego bricks. JavaScript is like having a huge pile of lego pieces, and you need to figure out how to assemble them into a house from scratch. React, on the other hand, gives you a set of pre-designed lego components (like doors, windows, and walls) that you can easily snap together to build your house. This makes building

and maintaining the house much simpler and less error-prone.

So, React is like a set of tools and rules that help you build web interfaces more efficiently and maintain them as they grow, without starting from scratch every time.

React's declarative approach

Think of React's declarative approach as writing a to-do list for a personal assistant. In an imperative task delegation, you'd specify detailed instructions for each task, like, "Go to the store, buy milk, check the expiry date, and return home." You're managing every task step by step.

With React, you act like a project manager. You create a high-level task list, saying, "I need groceries from the store," and your assistant handles the logistics, deciding the best route, checking for discounts, and ensuring everything is brought back. You're focused on what needs to be done, not micromanaging the process.

In this task delegation analogy, React allows you to outline the desired tasks (UI state) without getting bogged down in the specifics (DOM updates). When your shopping list (data) changes, React efficiently manages the execution of tasks (UI updates), making sure your web app stays organized and efficient. It's like having a reliable assistant who takes care of

the details, making your coding more efficient and your web app management hassle-free.

To set up a react project using vite check this out:

<https://www.digitalocean.com/community/tutorials/how-to-set-up-a-react-project-with-vite>

Creating and rendering components

What are Components?

Components in React can be thought of as building blocks for building user interfaces. They are like reusable pieces of a web page that you can create and customize, and then you can put these pieces together to create a complete web application.

Imagine you are building a web page, and you want to break it down into smaller, manageable parts. Each part could be a component. For example, you might have a component for the header of your page, another for a navigation menu, one for a product list, and so on.

In React, components are defined as JavaScript functions or classes that return JSX, describing what should be rendered on the screen.

Creating a Component:

React component names must start with a capital letter.

There are two common ways to create a component: as a functional component or as a class component.

1. Functional Component :

Functional components are simpler and more concise. These are simply like JavaScript functions.

2. Class Component :

These components are simple classes (made up of multiple functions that add functionality to the application). All class components are child classes for the Component class of ReactJS. Class components are used when you need to manage component state or use lifecycle methods.

What do you mean by rendering a component?

Rendering a component in React refers to the process of displaying or showing a component's user interface on a web page. In simpler terms, it means making the content and functionality defined within a React component visible to the user in a web application.

When you render a component, you are instructing React to take the component's description (typically written in JSX) and transform it into actual HTML elements that appear on the screen.

To render a component:

include the file in the header of "App.js" and pass the props*.

Syntax :

```
<Component_name prop="value" />
```


Props

In React, "props" is a shorthand for "properties," and they are a way to pass data from a parent component to a child component. Props allow you to customize and configure child components, making them dynamic and reusable.

Props are read-only, meaning that child components cannot modify the data they receive via props. They are purely for receiving and using data.

React Props are like function arguments in JavaScript and attributes in HTML.

How do you pass props between components?

Parent (P): You have a bag of candies, and you want to give some to your child.

Pass Props: To pass candies, you put them in a small box (props), and you write your child's name (prop name) on the box. Then you hand the box to your child.

Child (C): Your child receives the box with their name on it and opens it to find the candies (props) inside.

In this example, the candies are like data, and the box with your child's name on it is like a prop. You use props to give information from the parent component to the child component in React.

States

States in React

Imagine you're building a digital thermostat. The "state" in React is like the current temperature displayed on the screen. It's a way for your app to remember and show information that changes over time. For instance, as the room gets warmer or cooler, the displayed temperature updates to show the current temperature.

In React, we use "state" to make sure our app can remember and show changing information like this. It's like a note that React keeps, and whenever something important changes (like the temperature), React takes a look at the note and updates what's displayed on the screen to match.

The state object is where you store property values that belong to the component. Whenever the state object changes, the component re-renders.

useState

When you want to change something on your web page (like updating a score in a game or displaying a new message), you need two things:

Remember the Change: You need a way to remember what you want to change. For instance, if you're playing a game, you need to remember the score.

Make the Change Visible: You also need a way to show the change on your web page. If your score goes up, the web page should update to show the new score.

The `useState` Hook* in React helps with both of these things:

Memory Box: It gives you a special "memory box" where you can put the thing you want to change (like the score in a game). This box remembers it even when your web page refreshes or updates.

Magic Button: It also gives you a special "magic button" that you can press when you want to make the change visible.

When you press this button, React looks in the memory box, sees what's changed, and updates your web page to show the new information. So, if your score goes up, React makes sure the new score is displayed.

Hook.*

[*Hooks are like building blocks in React that allow you to use various features in your web applications. These building blocks can be either pre-made (built-in) by React or custom-made by you by combining them. *(We'll discuss about this in detail later)*]

Prop Drilling

Think of prop drilling like passing a message along a line of people. Imagine you're playing a game of "telephone" with your friends, and you want to tell the last person in line a secret message. You whisper it to the first person, who whispers it to the next, and so on until it reaches the last person.

"Prop drilling" is a term used in React to describe the process of passing data from a parent component to a deeply nested child component through intermediary components. It can occur when you have a component hierarchy where data needs to be passed down multiple levels.

While prop drilling works, it can lead to several challenges:

Complexity: As your component tree deepens, prop drilling can make your code more complex and harder to maintain.

Performance: Passing props through multiple components, especially when those components don't use the props themselves, can affect performance.

Readability: Code readability can suffer as you pass props through multiple levels of components.

EXTRA QUESTIONS:

Here are some questions you can use to deepen your understanding of States and Prop Drilling:

1. How is state different from props in React?
2. Develop a Toggle component that displays a button. Clicking the button should toggle the text between "ON" and "OFF" using state.
3. Construct a user profile page with three components: "UserProfile" , "UserDetails", and "UserLook". Pass user data from UserProfile to UserDetails and UserLook using props.

Conditional Rendering

What is conditional rendering?

Conditional rendering refers to the practice of displaying different content or components based on certain conditions or criteria. It allows you to control what gets rendered in the user interface (UI) based on the state of your application, user interactions, or any other condition you define.

Why conditional rendering?

Dynamic Content: Conditional rendering lets you display different content or components based on specific conditions.

User Experience: It enhances user experience by showing or hiding elements, handling errors, and offering responsive interfaces.

Performance: Helps optimize performance by rendering only what's needed, reducing unnecessary updates.

Authentication: Used to control access to parts of an app based on user authentication.

Error Handling: Allows for displaying error messages or components when issues occur.

Multi-Platform Support: Useful for creating responsive designs for different devices.

Conditional Rendering Techniques:

- if Statements
- Ternary Operator (? :)
- Logical Operators (&& and ||) etc.

Component Lifecycle

Components have a lifecycle that consists of different phases. Each phase has a set of lifecycle methods that are called at specific points in the component's lifecycle. These methods allow you to control the component's behavior and perform specific actions at different stages of its lifecycle.

There are typically three main stages in a component's lifecycle:

1. Mounting Stage: This is when a component is created and added to the DOM. It consists of three lifecycle methods:

- constructor: Component initialization.
- render: Rendering the component's UI.
- componentDidMount: Executed after the component is added to the DOM. It's often used for initial data fetching or setting up event listeners.

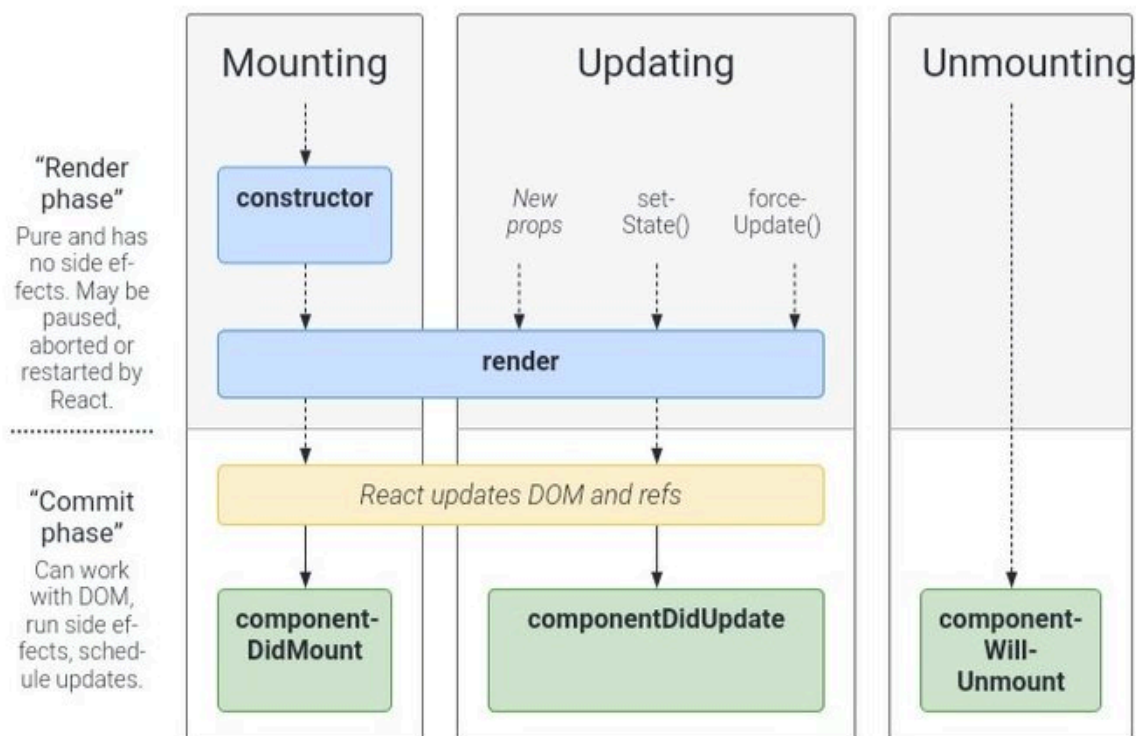
2. Updating Stage: This stage occurs when a component re-renders due to changes in its props or state. Key lifecycle methods in this stage include:

- shouldComponentUpdate: Allows you to optimize rendering by deciding whether or not the component should update.

- render: Re-rendering the component's UI.
- componentDidUpdate: Executed after the component re-renders. It's often used for tasks like updating the DOM or fetching new data.

3. Unmounting Stage: This is the stage where a component is removed from the DOM. The primary method used in this stage is:

- componentWillUnmount: Executed just before the component is removed. It's used for cleaning up resources like event listeners to prevent memory leaks.



Hooks

Hooks in React are like special tools that make it easier for components (the building blocks of React applications) to do things like remembering information, doing tasks at the right time, and sharing data with other parts of the app.

Hooks let us use different React features from our components. we can either use the built-in Hooks or combine them to build your own(custom hooks).

There are different kind of hooks:

- State Hooks
- Context Hooks
- Ref Hooks
- Effect Hooks
- Resource Hooks
- Performance Hooks
- Your own custom hooks.

State Hooks

Think of "state" in a component like a sticky note or a piece of paper that the component can use to remember important information. For instance, if you have a form on a website, the sticky note can hold the text you type into the form fields.

If you have a gallery of pictures, the sticky note can remember which picture you've selected to display. It's a way for the component to keep track of what's happening or what the user has done.

To add state to a component, we can use:

`useState` - declares a state variable that you can update directly.

`useReducer` -declares a state variable with the update logic inside a reducer function.

Context Hooks

Think of "context" as a messenger that allows a component to get information from its parent, even if there are many layers of other components in between.

For example, imagine your top-level component has information about the current style or theme of your app, like whether it's light or dark mode. Instead of passing this information as a prop through all the intermediate components, you can use context to send this message directly to the component that needs it, even if it's far down the component tree. It's like a shortcut for sharing important data without making a long chain of deliveries through props.

useContext - reads and subscribes to a context.

Ref Hooks

Think of "refs" as a way for a component to have a secret note that it keeps hidden, which contains some information it doesn't want to show on the screen. This information could be something like a reference to a specific part of a web page (like a button), or a timer that the component is using.

Unlike the regular information a component uses (which makes it update and change), this secret note (the ref) doesn't make the component change or re-render when you update it. It's like a special tool you can use when you need to work with parts of your app that don't follow the usual rules of React, like certain browser features.

In a way, refs are like a hidden escape route from the usual React way of doing things, allowing you to interact with the outside world when necessary, without causing your component to refresh or redraw.

useRef -declares a ref. You can hold any value in it, but most often it's used to hold a DOM node.

Effect Hooks

"Effects" are like connectors that help a component talk to and work smoothly with things outside of its own world.

These can be things like the internet (network), elements on a web page (browser DOM), animations, or even parts of your app that use a different set of rules for how they work (like widgets from a different user interface library).

So, when you want your component to interact with these external things, you use effects. They allow your component to understand and work together with the outside world, whether it's fetching data from the internet, animating elements, or making sense of non-React code. Effects help your component synchronize with these external systems, making your app more dynamic and responsive.

`useEffect` - connects a component to an external system.

eg:

useState Hook

`useState` is a built-in React Hook that lets you add a state variable to your component.

Usage of `useState`:

1. Adding State to a Component:

- Employ `useState` to introduce and manage state within your component.

2. Updating State Based on the Previous State:

- Utilize the previous state value to calculate and set the new state, ensuring accurate updates.

3. Updating Objects and Arrays in State:

- Manage complex state structures like objects or arrays, modifying specific elements as needed.

4. Avoiding Recreating the Initial State:

- Prevent unintentional state resets by using the functional update form to build upon the existing state.

5. Resetting State with a Key:

- Implement state resets by using a unique key or identifier to revert to initial values.

6. Storing Information from Previous Renders:

- Capture and retain information from previous render cycles, enabling data persistence and comparison.

useEffect Hook

useEffect is a built-in react hook that lets you synchronize a component with an external system.

Usage of useEffect:

1. Connecting to External Systems:

- Use `useEffect` to interact with external systems like APIs or databases.

2. Custom Hooks for Effects:

- Create custom hooks to encapsulate and reuse `useEffect` logic.

3. Non-React Widget Control:

- Manage non-React components or widgets within your React app using `useEffect`.

4. Data Fetching:

- Initiate data fetching operations with `useEffect` to retrieve and display data.

5. Reactive Dependencies:

- Specify dependencies in `useEffect` to trigger effects when specific values change.

6. Updating State Sequentially:

- Update state within `useEffect` based on previous state for sequential operations.

7. Optimizing Object Dependencies:

- Efficiently manage dependencies in `useEffect` to avoid unnecessary updates.

Custom Hooks:

There are so many built-in hooks in react, but still if you wish to create one for yourself you can use "Custom hooks".

You can make a custom hook by following these steps:

Start with a Function:

Begin by creating a regular JavaScript function. Custom hooks should have names that start with "use" to adhere to the hook naming convention.

Define the Hook:

Inside your custom hook function, you can use built-in hooks or other custom hooks if needed. These hooks can include `useState`, `useEffect`, or other custom hooks you've created.

Encapsulate Logic:

Identify the logic you want to encapsulate and abstract into the custom hook. This logic can involve state management, data fetching, event handling, or any other functionality you want to reuse across components.

Return Values:

Your custom hook should return values that components can use. Typically, this includes variables, functions, or both, depending on the functionality of your hook.

Use the Custom Hook:

Import and use the custom hook in your components, just like you would with built-in hooks. This allows you to reuse the encapsulated logic across different parts of your application.

Custom hooks are a powerful tool in the React developer's toolkit. They enable you to encapsulate and share stateful logic, promoting reusability and maintainability in your codebase. By creating custom hooks, you can streamline your components, separate concerns, and write cleaner, more efficient code.

useState

Firstly, import the useState from React:

```
import { useState } from 'react';
```

Syntax for useState:

```
//useState syntax:  
const [state, setState] = useState(initialState);
```

When your component is initially rendered, state will be set to the value of initialState. This sets up the initial state for your component. Whenever you want to update the state variable, you can call setState with a new value. For example, if you want to update state to a new value, you would do something like setState(newValue);. React will then re-render your component with the updated state, causing any relevant changes in your component's UI based on the new state.

useContext

Before we dive into the useContext hook, let's understand the problem it aims to solve. In a typical React application, you may have a component tree with several levels of nesting. Passing data from a parent component to a deeply nested child component can be cumbersome and lead to prop drilling, where props are passed down through multiple intermediate components. This can make your code harder to maintain and debug.

The useContext hook addresses this problem by providing a way to access data at any level in your component tree without the need for prop drilling.

useContext is a powerful React Hook that simplifies the process of reading and subscribing to context data from within your functional components. It promotes clean and efficient state management by eliminating the need for prop drilling and ensures that your components stay up-to-date with the latest context changes.

const value = useContext(SomeContext)

SomeContext refers to a context object that we have previously created using the React.createContext() function.

Context objects are created to hold and share data across components without the need for explicit prop passing.

useRef Hook

useRef allows us to create a reference to a value or object. This value can be of any type - it doesn't have to be a primitive like a string or number; it can also be an object, function, or even a DOM element.

When we use useRef to store a value, changing that value does not trigger a re-render of the React component. This is in contrast to using useState or useEffect, where changing the state or dependencies typically causes the component to re-render.

There are situations in a React component where you need to keep track of values that are important for some functionality but do not affect what gets displayed in the UI. These values might be related to maintaining component state, tracking measurements or positions of DOM elements, or managing subscriptions and timers.

const ref = useRef(initialValue)

initialValue is the initial value that you want to assign to the ref. This can be any value, including primitive values like numbers or strings, objects, functions, or even null. The

initial value you provide is typically the value you want to associate with the ref when the component first mounts.

Nested Routes

Think of nested routes as a way to organize the pages or sections of a website or web application. It's like having folders within folders on your computer to keep your files organized. In web development, nested routes are like "folders" of web pages that are contained within other web pages.

Nested routes are helpful for several reasons:

Organization: Just like you organize your files into folders, nested routes help developers organize their web pages. This makes it easier to find and work on specific parts of a website or app.

User Experience: Nested routes make it easier for website visitors to navigate. It's like having a clear menu structure so users can easily find what they're looking for.

Complexity: If a website or app has many features and pages, nested routes help manage this complexity by breaking it into smaller, more manageable parts.

Why do developers prefer nested routing?

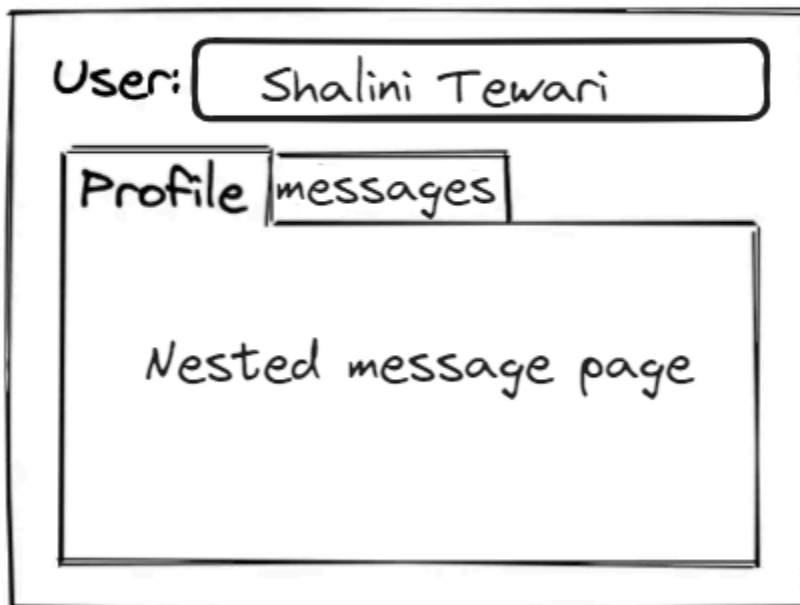
- *Code Organization*: Nested routers help organize code in a structured manner. They allow developers to modularize different sections of a web application, making it easier to manage and maintain. Each section of the application can have its own router, reducing code complexity.
- *Component Modularity*: Nested routers align with the principle of component-based development. Each route can be associated with a specific component, making it easier to understand and manage the codebase. Developers can focus on building and testing smaller components in isolation.
- *User Experience*: Nested routers improve the user experience by creating a logical and intuitive navigation structure. This is especially important for applications with multiple sections or features. Users can easily understand how to navigate through the application.
- *Flexibility*: Nested routers provide flexibility in defining complex user interfaces. Developers can nest routes within routes to create intricate, multi-level navigation systems. This flexibility is valuable for building dynamic and feature-rich applications.

- *Scaling Applications*: For larger and more complex web applications, nested routers are essential. They allow developers to break down the application into manageable pieces, making it easier to add new features or scale the application as it grows.

- *Testing*: Nested routers make it easier to test different parts of the application independently. This is valuable for ensuring that each section of the application functions correctly and does not interfere with other parts.

Eg for Nested routing

/user/messages



 @maybeshalinii

Extra Questions

Props:

1. What are props in React, and how do you pass data from a parent component to a child component using props?
2. Explain the difference between props and state in React.
3. Can you modify the value of props from inside a child component? Why or why not?
4. How would you handle default or missing props in a React component?
5. What is prop drilling, and how can you avoid it in your React application?

Components:

6. Describe the difference between functional components and class components in React. When would you use one over the other?
7. What is a higher-order component (HOC) in React, and why might you use it?

8. Explain React fragments and their use cases.
9. How do you create reusable components in React? Provide an example.
10. What is a controlled component in React, and why is it important?

State:

11. Describe the difference between "useState" and "useEffect" hooks in React.
12. How can you share state between sibling components in React?
13. Explain the concept of local component state versus global state management in React.

Routes:

14. How do you implement client-side routing in a React application?
15. What is the purpose of the React Router library, and how do you define routes in a React application using it?

16. How can you pass data between different routes in a React application?

Context API for State Management

Context API is a state management tool in React, a popular JavaScript library for building user interfaces. It provides a way to share data between components without having to pass props down through the component tree explicitly.

Context API is often used when you need to manage global state or share data that is used by multiple components in your application.

Overview of how Context API works for state management in React:

1. Creating a Context:

You start by creating a context using the `React.createContext()` function. This creates a context object that has two components: Provider and Consumer.

2. Provider Component:

The Provider component is used to wrap the part of your component tree where you want to share data. It accepts a value prop, which can be any JavaScript value (e.g., an object, a string, a function, etc.). The data you provide as the value will be made available to all child components that are consumers of this context.

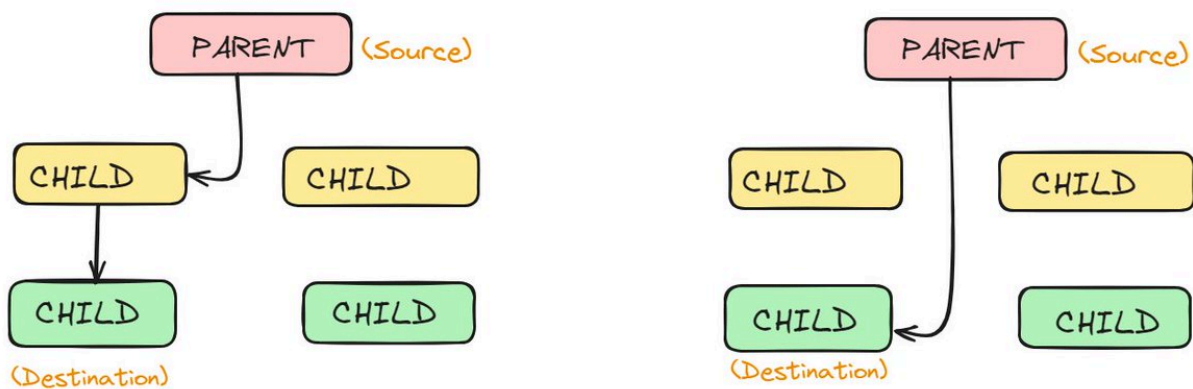
3. Consumer Component:

To access the data provided by the Provider, you can use the Consumer component or the useContext hook. The Consumer component uses a function as its child, which receives the value from the Provider.

4. Consuming Context Data:

You can consume the context data in any child component within the scope of the Provider. React will automatically update the component whenever the context data changes.

Passing Props from Parent to Child Vs How Context API Works



How to Get Started with the Context API

```
import { createContext } from 'react';  
export const ContextIs = createContext("");
```

we're importing createContext from React and using it to create a new context object.

```
function MyComponent() {  
  const data = "Hello, Fam!";  
  return (  
    <MyContext.Provider value={data}>  
      /* Child components */  
    </MyContext.Provider>  
  );  
}
```

Provider Component

/*child components*/ comment indicates where you would typically place the child components that need access to the data provided by the context. You would render your child components inside this Provider component to give them access to the context's data.

```
function MyChildComponent() {  
  return (  
    <MyContext.Consumer>  
      {(value) => <div>{value}</div>}  
    </MyContext.Consumer>  
  );  
}
```

Consumer Component

When you wrap your child components within the <MyContext.Provider>, any of those child components can access the data value provided by the context using the MyContext.Consumer component or the useContext hook. This allows you to share data across different parts of your React application without the need for prop drilling (passing data through multiple levels of components).

Introduction to Redux

What is Redux?

Redux is an open-source JavaScript library commonly used in front-end web development for managing the state of an application. It is often used in conjunction with popular JavaScript frameworks like React, Angular, and Vue, but it can also be used with other libraries and frameworks.

Redux was created to address the challenge of managing complex application states and their transitions in a predictable and maintainable way. It enforces a unidirectional data flow, which helps in debugging and understanding how data changes over time.

Key concepts and components of Redux:

- *Store*: Think of it as a big container that holds all the important data for your app. It's like a storage room where you keep everything you need.
- *Actions*: Actions are like requests or instructions that you send to the storage room (the store). For example, you might send an action to add something to your storage room or take something out.

- *Reducers*: These are like the workers in the storage room who follow the instructions (actions) and make changes in the storage room. They decide how the storage room should be organized based on the actions.

- *Dispatch*: Dispatching an action is like telling the workers to follow a specific set of instructions. You say, "Hey, workers, here's what I want you to do," and they do it.

- *Selectors*: These are like special tools you use to get exactly what you need from the storage room. Instead of searching through the entire room, you use a tool to find the specific item you want.

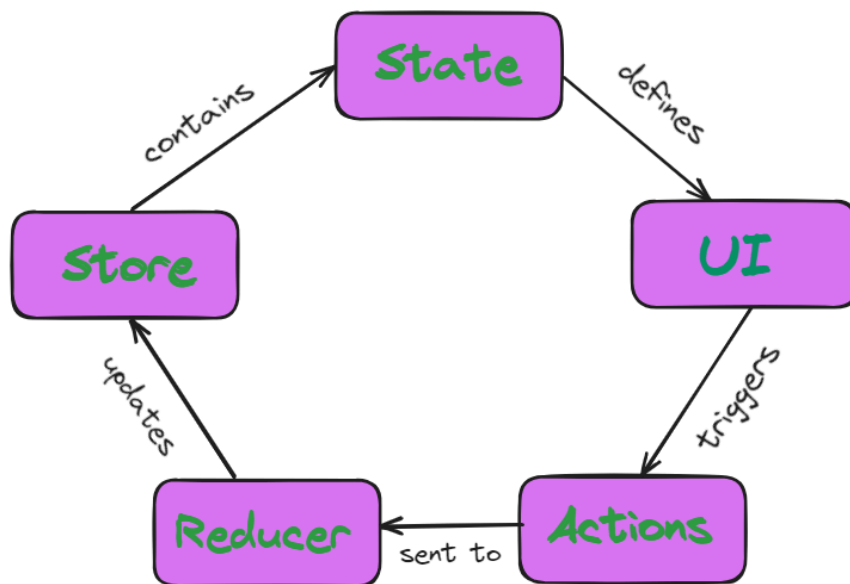
- *Middleware*: Middleware is like a security guard or a manager. It can do things before or after an action is completed. For example, it can check if the action is allowed or perform additional tasks.

React Redux includes a `<Provider />` component, which makes the Redux store available to the rest of your app. Also, it provides a pair of custom React hooks that allow your React components to interact with the Redux store.

To use React Redux with your React app, install it as a dependency.

`npm install react-redux`

Introduction to Redux



Getting the hang of Redux all at once can be a bit tricky. This quick overview offers a basic idea, but to truly dive into Redux, it's like needing more time for a friendly chat and a separate session to really get the hang of it. I advise you all to go and explore Redux, any questions are welcome here.

Learn more at :

<https://www.freecodecamp.org/news/redux-for-beginners/>

Fetching data from APIs

First of all, why do we need to fetch data from apis?

Fetching data from APIs is like getting information from a specialized messenger. Think of an API as a messenger that can go to different places and bring back specific information for you. Here's why we use these messengers:

Accessing Special Data: Sometimes, the information we need is stored in special places like other websites or databases. APIs help us get that information easily.

Getting Updates: If you need to know something right now or whenever it changes, APIs can get you the latest info quickly.

Saving Time: Instead of building everything from scratch, we can use APIs like pre-made tools to make our own work faster and easier.

Keeping Things Safe: These messengers also make sure only the right people can get the information, so it's safe and private.

Putting Things Together: You can use these messengers to combine information from different places, like making a recipe with ingredients from various stores.

Making Apps Better: Apps use APIs to add cool features, like showing the weather, finding nearby restaurants, or logging in with your social media account.

So, APIs are like messengers that help us bring in special data, save time, and make our applications more useful and powerful.

Two common choices to fetch data:

1. *fetch()*: A built-in JavaScript function for making HTTP requests.
2. *axios*: A popular third-party library for making HTTP requests.

fetch()

The `fetch()` function is a built-in JavaScript method for making HTTP requests to fetch resources, such as data or files, from a specified URL or endpoint. It is a fundamental part of modern web development and is often used to interact with APIs to retrieve data in web applications.

Syntax:

fetch(input[, init]);

-*input*: A URL or a string representing the URL from which to fetch the resource.

-*init (optional)*: An optional configuration object that allows you to customize the request, including headers, method, and more.

Axios

Before using Axios, you need to install it in your project.

npm install axios

Axios allows you to make GET requests to retrieve data from a specified URL. You can also make POST requests to send data to a server.

fetch

```
// Create a function to fetch data from the API
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data'); // Replace with your API endpoint
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();
    console.log(data); // Handle the data here
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}

// Call the fetchData function when needed, e.g., in a component's useEffect
useEffect(() => {
  fetchData();
}, []);
```

Axios

```
import axios from 'axios';

// Create a function to fetch data from the API
async function fetchData() {
  try {
    const response = await axios.get('https://api.example.com/data'); // Replace with your API endpoint
    console.log(response.data); // Handle the data here
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}

// Call the fetchData function when needed, e.g., in a component's useEffect
useEffect(() => {
  fetchData();
}, []);
```

Handling Errors in React Components

React has become one of the most popular JavaScript libraries for building user interfaces. While React's declarative nature and component-based architecture make it powerful and efficient, handling errors in React components can be challenging.

1. Error Boundaries:

React introduced the concept of error boundaries, which are special components that catch errors during rendering, in lifecycle methods, and during the constructor. To create an error boundary, define a component with the "componentDidCatch" method. This method receives two arguments: the error and the error information. Error boundaries enable you to gracefully handle errors without crashing the entire application.

Imagine you're building a website with React, and sometimes things go wrong, causing parts of your website to crash and disappear from the screen. To avoid this, React gives you a way to protect those parts by putting them inside something called an "error boundary."

An error boundary is like a safety net for your website. When an error happens inside it, instead of making that part vanish,

you can show something else, like an error message or a friendly "Oops, something went wrong" notice. It helps keep your site looking good even when there are issues.

componentDidCatch(error, errorInfo){}

Alternatives:

- When you're working on a React application, you often create pieces of your app called components. Originally, React allowed you to make components in two different ways: as classes or as functions. However, the newer and more common way is to use functions.

So, if you have an older component made as a class, it's usually a good idea to change it into a function instead. This makes your code simpler and more in line with modern React practices. It's like upgrading your component to a more modern version.

- When you're working on a React component, you might have built it using a class and used special functions called "lifecycle methods" to make it work. But nowadays, the trend is to use function components instead of classes.

So, if you have an old component that relies on these lifecycle methods because it's a class, you might want to upgrade it to a function component. This means you'll

rewrite it using functions instead of classes and use modern techniques for managing its behavior. It's like giving your component a makeover to make it more up-to-date.

2. Error Handling Components:

Create reusable error handling components that encapsulate error display logic. These components can be used throughout your application to display error messages consistently. This approach promotes code reusability and maintains a clean separation of concerns.

How would you handle errors in function based components?

- Use Try-Catch Blocks
- Display Error Messages
- Using Error Boundaries

Extra exercises:

Here are some exercises that should help you solidify your understanding of React concepts and best practices.

Create a Functional Component:

Question: How do you create a functional React component?

Task: Create a simple functional React component that renders "Hello, World!" on the screen.

State Management:

Question: How do you manage state in a functional component?

Task: Convert your component from the previous exercise into a stateful component using the useState hook.

Implement a counter that increments when a button is clicked.

Effect Hook:

Question: How can you perform side effects in React components?

Task: Add the useEffect hook to your component and use it to change the document title when the component mounts.

Props and Component Composition:

Question: How do you pass data from a parent component to a child component?

Task: Create a parent component that passes a message as a prop to a child component. The child component should render the message.

Routing with React Router:

Question: How can you set up routing in a React application?

Task: Install and configure React Router in your project. Create two separate routes with different components, such as a Home page and an About page. Use `<Link>` components to navigate between them.

Fetch Data from API:

Question: How do you fetch data from an API in React?

Task: Create a component that fetches and displays data from a public API of your choice, such as a list of posts or weather information.

Form Handling:

Question: How can you handle form input in React?

Task: Build a form component that allows users to enter a name and submit it. Manage the form state and display a greeting message with the submitted name.

Conditional Rendering:

Question: How do you conditionally render components or content in React?

Task: Create a component that conditionally renders content based on a boolean state. For example, show a loading spinner while data is being fetched and then display the fetched data.

Local Storage Interaction:

Question: How can you interact with local storage in React?

Task: Modify a component to save and retrieve data (e.g., a to-do list) from local storage. Allow users to add and remove items.

Custom Hook:

Question: What are custom hooks in React, and how do you create them?

Task: Create a custom hook that encapsulates a specific piece of logic (e.g., form validation or API requests). Use this custom hook in a component to demonstrate its reusability.

Debouncing in react

What is debouncing?

Debouncing is a technique used in React (and other JavaScript applications) to optimize the performance of functions that are triggered frequently, such as event handlers or search input handlers. It helps prevent unnecessary and rapid execution of a function, which can lead to performance issues.

Why Debouncing?

1. Performance Optimization: One of the primary reasons for using debouncing is to optimize the performance of your application. When you have event handlers, such as those for input fields, that trigger frequently (e.g., while typing or resizing a window), executing the associated functions on every event can be resource-intensive and lead to poor performance. Debouncing ensures that these functions are executed only after a certain delay or when the user pauses their action, reducing the overall number of function calls and improving performance.

2. Reducing API Calls: In scenarios where you need to make API requests based on user input, like search suggestions,

debouncing helps reduce the number of unnecessary API calls. Without debouncing, an API request would be sent for every keystroke, potentially overwhelming your server with requests. Debouncing ensures that the request is made only when the user has finished typing or after a specified delay, resulting in a more efficient use of resources.

3. *Smooth User Experience:* Debouncing can provide a smoother and more user-friendly experience by preventing rapid and jarring updates to the UI. For example, if you're implementing a live search feature, debouncing allows you to update search results after the user has finished typing, providing a more coherent and less distracting experience.

4. *Preventing Unwanted Interactions:* In some cases, you might want to prevent unwanted interactions or side effects caused by rapid user input. Debouncing can help in situations where you want to ensure that a particular action, like submitting a form or saving data, only occurs once the user has stopped interacting with the UI element.

5. *Resource Efficiency:* By delaying the execution of functions, debouncing can help save computational resources and battery life on mobile devices. It ensures that the CPU is

not constantly engaged in processing events and allows it to enter lower power states during periods of inactivity.

Debouncing using lodash

The code imports the debounce function from the "lodash" library.
This function allows you to create a debounced version of a function.

```
import { debounce } from "lodash";  
  
// Create a debounced function called handleChangeWithLib  
const handleChangeWithLib = debounce((value) => {  
  // Make an API request to an example endpoint with the provided search value  
  fetch('https://jsonplaceholder.typicode.com/posts?q=${value}')  
    .then((res) => res.json())  
    .then((json) => {  
      // Assume you have a function setSuggestions to update suggestions with the API response  
      setSuggestions(json);  
    });  
}, 500);
```

This function will be used to handle user input in a way that limits the frequency of API requests.
The debounced function takes one argument, value, which is typically the user's input.

The fetch request is wrapped in the debounced function, which means that it will only be executed if there is a pause of at least 500 milliseconds between calls to handleChangeWithLib. This prevents rapid API requests as the user types, optimizing the performance of your application.

Higher order components

What are Higher order components?

A higher-order component is a function that takes a component and returns a new component.

Or we can say that, A higher-order component (HOC) is like a helper function in React that makes it easier to reuse certain parts of your component code. It's not something built into React itself; it's just a technique or pattern that developers use because of the way React works.

Imagine you have a bunch of components in your app, and some of them share similar logic or behaviour. Instead of copying and pasting that same code into each component, you can create a higher-order component.

Here's how it works:


- You create a regular JavaScript function (the HOC) that takes one of your components as an input.
- Inside this function, you can add extra functionality, like handling data loading or setting up event listeners.

-Then, the HOC returns a new component that includes this extra functionality.

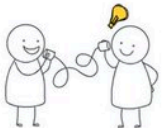
-Now, you can use this new component in your app just like any other component, and it automatically has the extra features you added with the HOC.

So, in simpler terms, a higher-order component is like a tool you use to easily share and reuse code among different components in your React app, making your code more efficient and maintainable.

In a nutshell, a higher-order component is a function that takes a component as its argument and returns a new component with additional props, behaviour, or other modifications. HOCs allow you to abstract and reuse logic and functionality that can be shared across multiple components. They are a way to achieve



Higher Order Component



```
// A higher-order component that adds a "loading" prop to a component
function withLoading(Component) {
  return function WithLoading(props) {
    if (props.isLoading) {
      return <div>Loading...</div>;
    } else {
      return <Component {...props} />;
    }
  };
}

// Usage of the higher-order component
const EnhancedComponent = withLoading(MyComponent);
```

A higher-order component named `withLoading` is defined. This HOC takes one argument, which is a React component (`Component`).

Here, the `withLoading` HOC is used to create a new component called `EnhancedComponent`. It does this by passing `MyComponent` as an argument to `withLoading`. As a result, `EnhancedComponent` now has the additional functionality provided by the `withLoading` HOC.

this code defines a higher-order component that can be used to wrap other components, adding a loading indicator based on the value of the `isLoading` prop. This is a simple example of how you can use HOCs to enhance and reuse component logic in a React application.

HOW TO STRUCTURE YOUR FRONTEND CODE IN REACT.

[with explanation]

```

├── src
│   |
│   ├── components
│   │   ├── Cards
│   │   │   ├── MainCards.jsx
│   │   │   └── Buttons
│   │   └── api
│   │       ├── Auth.js
│   │       ├── Event.js
│   │       └── Pages
│   │           ├── HomePage
│   │           │   ├── HomePage.jsx
│   │           │   └── LoginPage
│   │           │       └── LoginPage.jsx
│   │       └── contexts
│   │           ├── AuthContext.js
│   │           └── EventContext.js
│   └── hooks
│       ├── useAuth.js
│       └── useEvent.js
```



```
|_  utils
| |_  HelperFunctions.js
| |_  Date.js
|_  assets
| |_  images
| | |_  logo.svg
| | |_  background.jpg
| |_  styles
|   |_  global.css
|   |_  theme.js
|_  App.jsx
|_  index.js
```

Here is a brief explanation:

- **src:** This is the main folder where your React app lives.
- **components:** Think of this like a box of LEGO pieces. It holds small, reusable parts of your app, like building blocks. Inside, you have separate folders for different types of blocks:
 - Cards:** These are special blocks that you can use to create things like information cards.
 - Buttons:** Here, you keep different types of buttons you can use in your app.

- **api:** This is like a special toolbox where you keep tools for talking to the internet.

Auth.js: It probably has stuff for logging in and keeping your app secure.

Event.js: This could handle things like scheduling events or activities in your app.

- **Pages:** Imagine this as a storybook with different pages of your app.

HomePage: This page contains all the stuff for your app's main screen.

LoginPage: This is where users can log in to your app.

- **contexts:** Think of these like invisible helpers that carry messages between different parts of your app.

AuthContext.js: It helps parts of your app know if someone is logged in or not.

EventContext.js: This one helps your app share information about events.

- **hooks:** These are like shortcuts or tricks to make parts of your app work better.

useAuth.js: It helps with all the stuff related to logging in and out.

useEvent.js: This one makes working with events easier.

- **utils:** This is your toolbox of handy tools for your app.

HelperFunctions.js: Contains tools for doing all sorts of jobs in your app.

Date.js: It might help you handle dates and times in your app.

- **assets:** This is where you keep pictures, designs, and styles for your app.

images: Stores the pictures your app uses, like a logo or background.

styles: Holds the rules for how your app should look.

- **App.jsx:** Imagine this as the main stage where your app comes to life. It decides what to show and how everything should work.

- **index.js:** This is like the stage manager. It tells your app to get ready and appear on the screen for everyone to see.

This organized structure helps you build and manage your website in a neat and organised way, making it easier to understand and work on, especially when your project gets

bigger and more complex and you have to make changes very often.

I am not saying that this is the only or the best way, this is what I usually follow and thought about sharing it.

Concurrent React

With the introduction of Concurrent Mode in React, developers now have a powerful tool at their disposal to create even more responsive and efficient web applications.

What is Concurrent React?

Concurrent React is a set of new features in React designed to help apps stay responsive and gracefully adjust to the user's device capabilities and network speed. In simple terms, it's like React's way of ensuring that your web app continues to work smoothly, even when faced with complex and resource-intensive tasks.

Or we can say, concurrent React might sound complex, but at its core, it's not a visible feature you interact with directly in your code. Instead, it's like a secret worker behind the scenes that helps React get things done more efficiently.

Imagine you have a chef in a restaurant who can prepare multiple dishes at once without getting overwhelmed. That's what React does with your user interface. It's like having a chef who can work on many orders simultaneously.

React uses some smart strategies, kind of like the chef's recipe book, to manage this efficiently. It organizes tasks, deciding which ones to work on first, and stores some extra

information to avoid any delays. But as a developer, you don't need to worry about these behind-the-scenes details. You just benefit from a faster and more responsive user interface, which is what matters most.

Why is it essential?

- *Improved User Experience*: With Concurrent Mode, your app becomes more responsive and user-friendly.

Slow-loading components or data won't hinder the user experience as React can manage these operations efficiently.

- *Better Performance*: Concurrent React's ability to prioritize and split work into smaller pieces means that your app's performance remains consistent even during resource-heavy tasks.

- *Adaptability*: Concurrent Mode allows your app to adapt to varying network conditions and device capabilities, making it ideal for creating progressive web apps (PWAs) that can work on different devices and connections.

How to get started?

- Ensure you're using a version of React that supports Concurrent Mode.

- *Use **Suspense***: Familiarize yourself with the Suspense component and its usage to control data loading and rendering.
- *Optimize Components*: Split your components into smaller, more manageable parts. This makes it easier for React to prioritize and render them efficiently.

Suspense:

" <Suspense> lets you display a fallback until its children have finished loading. "

Which means, it is like a helper that makes dealing with things that take time, like getting data from the internet or loading parts of a website, much easier. It also makes sure your website doesn't look broken while it's waiting for these things to finish.

Think of it as a way to say, "Hey, while we're waiting for something to happen, show this message or picture to keep the user informed." So, it's like giving your users a "Loading..." sign instead of a blank screen, which is much friendlier.

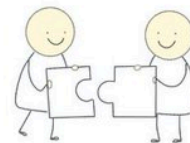
It's all about making your web apps smoother and more user-friendly, especially when they need to fetch data or do complex tasks in the background.

Key Aspects of Suspense:

- *Declarative Data Handling*: Easily load data within your components, improving code organization and reducing the need for complex data management.
- *Smoother User Experience*: Show loading indicators or user-friendly content while data loads, keeping your app responsive.
- *Error Handling*: Handle errors gracefully when data fetching goes wrong.
- *Simplified Data Coordination*: Manage the loading of multiple data pieces effortlessly, ensuring everything is ready before rendering a component.



Suspense



```
import React, { Suspense } from 'react';
```

```
const MyComponent = () => {
```

```
  return (  
    <Suspense fallback = {<div>Loading...</div>}>  
      <AsyncDataComponent />  
    </Suspense>  
  );  
};
```

Use the 'Suspense' component to wrap the part of your component that requires asynchronous data.

```
const AsyncDataComponent = () => {
```

```
  // Use a React.lazy() or other asynchronous data loading mechanism.  
  const data = loadAsyncData();  
  return <div>{data}</div>;  
};
```

An imaginary function for data loading.

```
export default MyComponent;
```

While the data is loading, the fallback content specified in the Suspense component (i.e., "Loading...") is displayed to the user.

Once the data is fetched, it is displayed within a <div> element.



Introduction to Server and Client side rendering

Server-side rendering (SSR) and client-side rendering (CSR) are two different approaches to rendering web applications, and they have their own advantages and use cases. *In the context of React, you can use both of these rendering methods.*

Server-Side Rendering (SSR):

- In SSR, the initial rendering of the web page occurs on the server before it is sent to the client's browser. The server generates the HTML content and sends it to the client, which can be displayed quickly as it arrives.
- React components are rendered on the server, and the resulting HTML is sent to the client.

SEO (Search Engine Optimization) benefits: Search engines can easily crawl and index SSR pages because the content is available in HTML form from the beginning.

- Faster initial load: SSR can provide faster initial page loads compared to CSR, especially on slow network connections or less powerful devices.

Imagine you're in a restaurant. With SSR, the chef (the server) prepares your meal completely in the kitchen (on the server) and brings it to your table (your web browser) as a fully cooked dish. You can start eating right away because it's ready.

Use SSR when you want your web page to appear quickly and be easy for search engines to understand. It's like getting a ready meal.

Use Cases for SSR:

1. Content-driven websites or blogs.
2. E-commerce sites.
3. Any application where SEO is crucial.
4. Sites that require fast initial rendering.

Client-Side Rendering (CSR):

In CSR, the initial HTML content is minimal, and JavaScript (including React) runs on the client-side to render the rest of the content. The client browser is responsible for rendering the UI.

CSR is often used in single-page applications (SPAs), where the initial load is relatively lightweight, and subsequent content updates are handled by JavaScript.

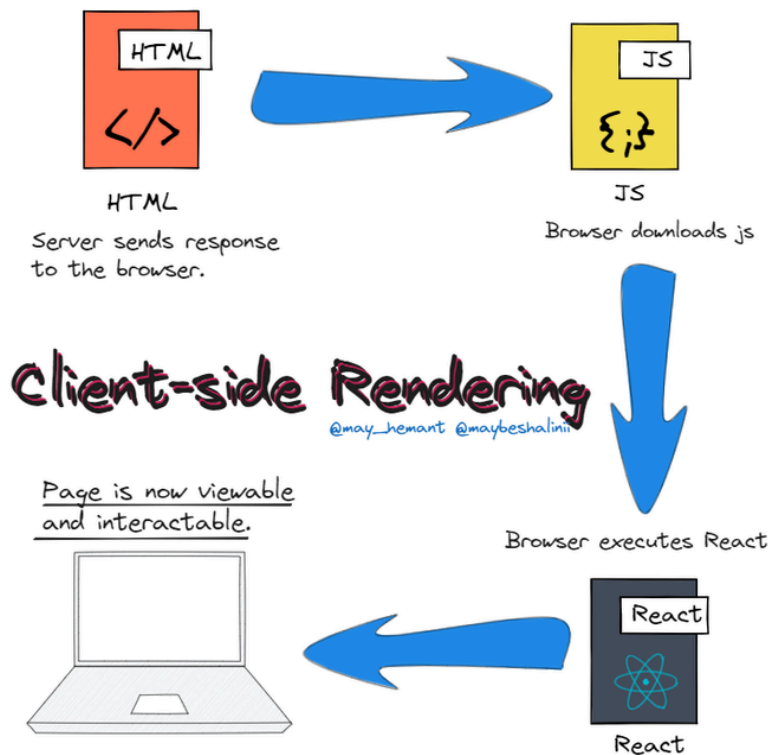
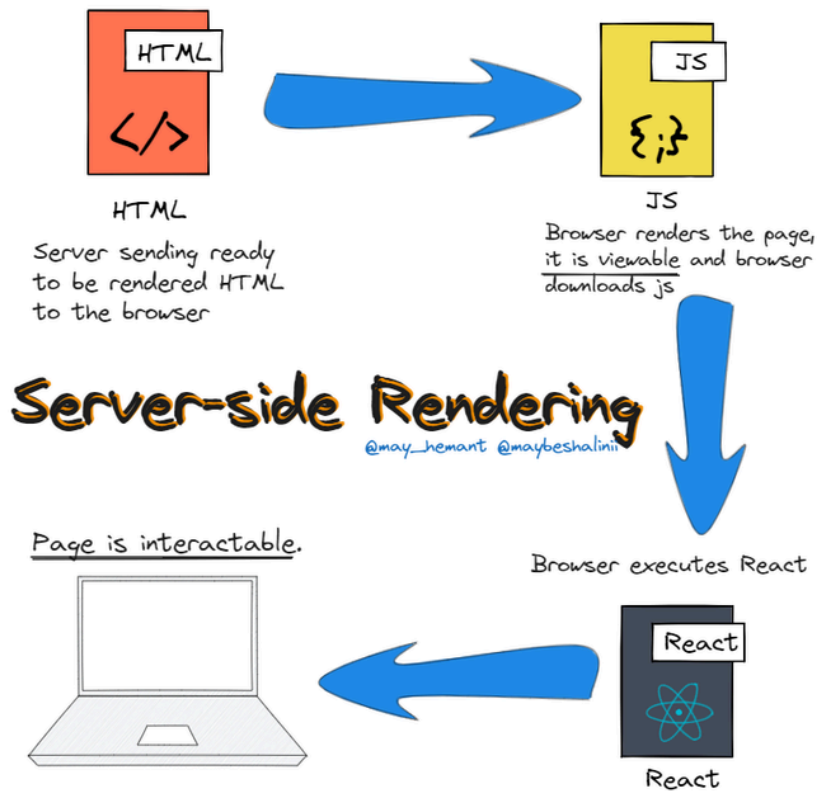
CSR is often associated with a faster and more interactive user experience after the initial load, as only the data needs to be fetched from the server, not the entire HTML structure.

Picture a different restaurant where the chef (the server) only brings you a plate with some ingredients (basic web page) and gives you a small kitchenette (JavaScript in your browser). You, the customer (your browser), have to put everything together and cook the meal (render the web page) before you can start eating.

Use CSR when you want your web page to be more interactive and can wait a bit for it to load initially. It's like cooking your own meal; it takes a little more time, but you have more control over it.

Use Cases for CSR:

1. Interactive web applications.
2. Dashboards and data-driven applications.
3. Applications that rely on real-time data updates.



Project Ideas (cause web dev is all about building projects)

10 Project ideas to strengthen your react concepts:

1. Online Quiz App:

- Plan the quiz application's structure and user interface.
- Create a React project and set up components for the quiz questions, timer, and score.
- Implement state management for the quiz data, user progress, and score.
- Allow users to select answers, track time, and display the final score.
- Optionally, include features like a Leaderboard or multiple quiz categories.

2. Task Management Dashboard:

- Define the structure for managing projects, tasks, and team assignments.
- Set up routing and create components for the dashboard, project view, and task management.
- Implement user authentication for team members.
- Allow users to create projects, assign tasks, set deadlines, and monitor progress.

- Use data visualization to show project status and task completion.

3. Music Player:

- Plan the features for your music player, including play, pause, skip, and volume control.
- Set up a React project and create components for the player interface.
- Implement audio handling to play music tracks.
- Allow users to control playback and adjust the volume.
- Create a playlist feature to manage and switch between songs.

4. Finance Tracker:

- Define the components needed to track income, expenses, and savings.
- Set up a React project and create components for income, expenses, and savings entries.
- Implement state management for financial data.
- Allow users to log income and expenses, categorize transactions, and set savings goals.
- Use data visualization to display financial trends and savings progress.

5. To-Do List App:

- Define the project requirements and create a design or wireframe.
- Set up a React project.
- Create components for the task list, task form, and individual tasks.
- Implement state management to handle task data.
- Allow users to add, update, and delete tasks.
- Style the app using CSS or a CSS framework.

6. Weather App:

- Define the project requirements and choose a weather API to use.
- Set up a React project.
- Create components for displaying weather information and user input.
- Make API requests to fetch weather data.
- Display the weather data to the user.
- Style the app and handle error cases gracefully.

7. E-commerce Website:

- Plan the website's structure, including product listings, shopping cart, and checkout.
- Set up routing for different pages (product listings, cart, checkout).

- Create components for product listings, shopping cart, and checkout.
- Implement state management for the shopping cart and product data.
- Allow users to add products to the cart, update quantities, and proceed to checkout.
- Implement a payment system or simulate one for checkout.

8. Blog or Portfolio Website:

- Plan the website's structure and design the layout.
- Set up a multi-page React application with routing.
- Create components for different sections of the website (e.g., home, blog, portfolio).
- Implement data retrieval for blog posts or portfolio items.
- Style the website and ensure a responsive design.
- Optionally, set up a back-end for content management.

9. Chat Application:

- Plan the chat application's architecture and user interface.
- Set up a React project and add real-time capabilities using WebSocket or a real-time database.
- Create components for the chat interface and user management.
- Implement user authentication.

- Allow users to send and receive real-time messages.
- Style the chat application.

10. Social Media Dashboard:

- Plan the dashboard's layout and functionality, including post management and follower tracking.
- Set up routing and create components for different sections of the dashboard.
- Implement state management for user data, posts, and followers.
- Allow users to create and manage posts, track followers, and view analytics.
- Style the dashboard with a focus on data visualization.

Each of these project ideas will help you explore different aspects of React, such as state management, user interface design, and interaction with various data sources or APIs.

As you work on these projects, you'll gain practical experience and build a strong foundation in React development.

Thank you, everyone, for your patience in waiting for this document. I am really glad I decided to post on Twitter (X) almost a year ago, as this community has given me so much love and appreciation. I hope you all will like my token of appreciation to y'all.

If you have any doubts, please reach out. I always try to reply to as many people as I can.

Thank you again for using my resources. I try to give the best I can. Happy coding, and let's grow together.

Your fellow developer,

Shalini.