Prop Drilling

- Prop drilling is a method where we pass a props with another component with the help of all the components that come between them.

- "For example, imagine that you are in school, sitting at the back of the classroom, while your best friend occupies the front row. Suddenly, your best friend realizes they need a pen, and you happen to have a spare one. In this situation, you can ask the friend sitting on the bench next to you to pass the pen forward, instructing them to continue passing it along until it reaches your best friend."

- To avoid all this, we use the Context API. It works like we called the teacher and requested him/her to give this pen to his friend who are sitting on the first bench.

Problems with Prop Drilling

- Becomes very difficult when we have a deeper level of nesting and we are passing the props.
- Useless re-renders will be there in case that prop is changed from the parent via an event.
- Unstable & Slow Application.

Solution of Prop Drilling

We can solve this problem with the help of below methods :-
- Using Context API.
- Using Redux and other global state management libraries.

Examples of Prop Drilling

```
import React,{useState} from 'react'

// FirstComponent
const App = () => {
  const [languageName, setLanguageName] = useState("ReactJS")
  return (
     <SecondComponent firstProp={languageName}/>
  )
}

// SecondComponent
const SecondComponent = ({firstProp}) => {
  return(
    <ThirdComponent secondProp={firstProp}/>
  )
}

// ThirdComponent
const ThirdComponent = ({secondProp}) => {
  return (
     <FourthComponent thirdProp={secondProp} />
  )
}

// FourthComponent
const FourthComponent = ({thirdProp}) => {
  return (
     <h1>{thirdProp}</h1>
  )
}

export default App;
```

ESLint

- ESLint is a widely used open-source tool for static code analysis in JavaScript-based projects, including React applications.
- It is not exclusive to React but is commonly used in React projects to ensure code quality, maintainability, and consistency.
- ESLint is highly configurable and can be customized to enforce specific coding styles and best practices.

Key Aspects of ESLint

Here are some key aspects of ESLint in React applications:

Static Code Analysis: ESLint performs static code analysis, meaning it checks your code without executing it. It can identify issues such as syntax errors, code style violations, and potential bugs.
Customizable Rules: ESLint provides a wide range of configurable rules and plugins that can be tailored to meet your project's specific requirements. You can enforce rules related to code formatting, best practices, and more.
Integration with Editors and Build Tools: ESLint can be integrated into various code editors and integrated development environments (IDEs) to provide real-time feedback to developers. It can also be integrated into build tools like Webpack or Babel to enforce code quality as part of the build process.
React-Specific Rules: ESLint has plugins and configurations tailored for React. These rules help ensure that your React components are written in a consistent and best-practice-oriented manner.
Preventing Common Errors: ESLint can help catch common mistakes in React, such as missing or unused variables, issues with prop types, or incorrect usage of hooks.

By using ESLint in your React project, you can maintain a high level of code quality and consistency across your codebase, making it easier to collaborate with other developers and catch potential issues early in the development process. It's considered a best practice to include ESLint in your React development workflow to improve code quality and maintainability.

Conditional Rendering

- Conditional rendering is a term to describe the ability to render different user interface (UI) markup if a condition is true or false.
- In React, "Conditional Rendering" allows us to render different elements or components based on a condition.
- The concept of Conditional Rendering applied often in the following scenarios: -
    - Rendering external data from an API.
    - Showing or hiding elements.
    - Toggling application functionality.
    - Implementing permission levels.
    - Handling authentication and authorization.

Methods of "Conditional Rendering"

a) By Using "if..else" statements

It is the easiest way to have a conditional rendering in React in the render method. It is restricted to the total block of the component. IF the condition is true, it will return the element to be rendered.

Example

```
import React,{useState} from 'react'
const App = () => {
  const [state, setState] = useState(true)
  const prntName = () => {
    if(state){
      return <h1>Hello ReactJS</h1>
    }else{
      return <h1>Hello NodeJS</h1>
    }
  }
  return(
    <>
      {prntName()}
    </>
  )
}
export default App;
```

b) By Using "ternary(?:)" operator

The ternary operator is used in cases where two blocks alternate given a certain condition. This operator makes your if-else statement more concise. It takes three operands and used as a shortcut for the if statement.

Example

```
import React,{useState} from 'react'
const App = () => {
  const [state, setState] = useState(true)
  return(
    <h1>{state ? "Hello ReactJS" : "Hello NodeJS"}</h1>
  )
}
export default App;
```

c) By Using "logical(&&)" operator

- This operator is used for checking the condition. If the condition is true, it will return the element right after &&, and if it is false, React will ignore and skip it.
- Short circuit evaluation is a technique used to ensure that there are no side effects during the evaluation of operands in an expression. The logical && helps you specify that an action should be taken only on one condition, otherwise, it would be ignored entirely.

Example

```
import React,{useState} from 'react'
const App = () => {
  const [state, setState] = useState(true)
  return(
    <h1>{state && "Hello ReactJS"}</h1>
  )
}
export default App;
```

d) By Using "Switch Case"

It is possible to have multiple conditional renderings. In the switch case, conditional rendering is applied based on a different state.

Example

```
import React,{useState} from 'react'
const App = () => {
  const [state, setState] = useState("React")
  switch(state){
    case "React":
      return <h1>Hello ReactJS</h1>
    case "React Native":
      return <h1>Hello React Native</h1>
    default:
      return <h1>Hello NodeJS</h1>
  }
}
export default App;
```

e) By Using "Nullish coalescing" operator

The nullish coalescing operator (??) is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand.

Example

```
import React,{useState} from 'react'
const App = () => {
  const [state, setState] = useState("React")
  return(
    <h1>Hello {state ?? ""}</h1>
  )
}
export default App;
```

f) By Using "Optional Chaining" operator

The optional chaining operator (?.) accesses an object's property or calls a function. If the object is undefined or null, it returns undefined instead of throwing an error.

Example

```
import React,{useState} from 'react'
const App = () => {
  const [languages, setLanguages] = useState({
    lang1:"ReactJs",
    lang2:"React Native",
    lang3:"NodeJS"
  })
  return(
    <h1>Hello {languages?.lang1}</h1>
  )
}
export default App;
```

Lists & Keys Rendering

- Lists are an important aspect within your app. Every application is bound to make use of lists in some form or the other.
- Lists are used to display data in an ordered format and mainly used to display menus on websites. In React, Lists can be created in a similar way as we create lists in JavaScript.

Keys in ReactJS

- Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity.
- A "key" is a special string attribute we need to include when creating lists of elements.
- Keys serve as a hint to React but they don't get passed to your component.

Example

```
import React from 'react'
const lst = ["HTML","CSS","JS","ReactJS"];
const App = () => {
  return lst.map((item,index) => <li key={index}>{item}</li>)
}
export default App;
```

Example

```
import React,{useState} from 'react'
const App = () => {
  const [data, setData] = useState([
    {id:1,name:"Rohit",email:"rohit@gmail.com"},
    {id:2,name:"Rahul",email:"rahul@gmail.com"},
    {id:3,name:"Rishabh",email:"rishabh@gmail.com"},
  ])
  return(
    <ul>
      {data && data.length !== 0 ? data.map((item) => (
        <li key={item.id}>Hello {item.name} your email id is {item.email}</li>
      )) : <li>No Data Found</li>}
    </ul>
  )
}
export default App;
```

Styling ReactJS App

- CSS in React is used to style the React App or Component.
- The style attribute is the most used attribute for styling in React applications, which adds dynamically-computed styles at render time.
- It accepts a JavaScript object in camelCased properties rather than a CSS string.

Methods for Styling Components

We have mainly four ways to style React Components that are listed below :-

a) Inline Styling
b) CSS ClassName / Stylesheets
c) Modular CSS
d) CSS In JS

a) Inline Styling

- The inline styles are specified with a JavaScript object in camelCase version of the style name.
- In Inline Styling we use the 'style' attribute in order to style React Components. The 'style' attribute accepts a JavaScript object with camelCase properties rather than a CSS string.

Examples

```
import React from 'react'
const App = () => {
  return(
    <h1 style={{color:"red",backgroundColor:"black"}}>Hello ReactJS</h1>
  )
}
export default App;
```

Examples

```
import React from 'react'
const App = () => {
  const txtColor = {
    color:"white"
  }
  const bgColor = {
    backgroundColor:"blueviolet"
  }
  return(
    <h1 style={{...txtColor,...bgColor}}>Hello ReactJS</h1>
  )
}
export default App;
```

b) CSS ClassName / Stylesheets

CSS 'ClassName' attribute is used to style component just like 'css classes'.

Examples

main.css

```
.hello{
    color: white;
    background-color: blue;
}
```

app.js

```
import React from 'react'
import "./main.css"
const App = () => {
  return(
    <h1 className='hello'>Hello ReactJS</h1>
  )
}
export default App;
```

c) Modular CSS

- CSS Module is another way of adding styles to your application.
- It is available only for the component which imports it, means any styling you add can never be applied to other components without our permission, and we never need to worry about name conflicts.
- We can create CSS Module with the .module.css extension like "[ComponentName].module.css".

Examples

App.module.css

```
.hello{
   color: white;
   background-color: blue;
}
```

app.js

```
import React from 'react'
import styles from "./App.module.css"
const App = () => {
  return(
    <h1 className={styles.hello}>Hello ReactJS</h1>
  )
}
export default App;
```

d) CSS In JS (homework)

- "CSS-in-JS" refers to a pattern where CSS is composed using JavaScript instead of defined in external files.
- This functionality is not a part of React, but provided by third-party libraries.

These libraries includes :-

- Styled Component
- Emotion
- Glamorous
- Radium