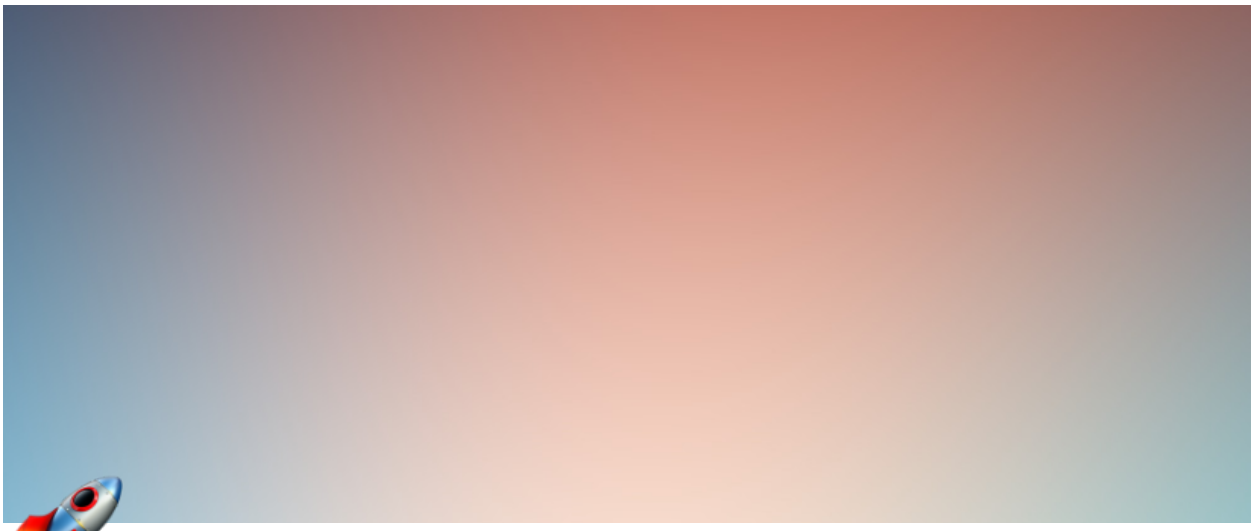

Episode-11 | Data is The New Oil

💡 Please make sure to follow along with the whole "**Namaste React**" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React dvelopment.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-11** first. Understanding what "**Akshay**" shares in the video will make these notes way easier to understand.

Q) What is `prop drilling` ?

> In React, `prop drilling` refers to the process of `passing down props (short for properties) through multiple layers of nested components`. This happens when a piece of data needs to be transferred from a higher-level component to a deeply nested child component, and it must pass through several intermediary components in between.

💡 Here's a simple example to illustrate prop drilling in React:

```
// Top-level component
function App() {
  const data = "Hello, prop drilling!";

  return (
    <div>
      <ParentComponent data={data} />
    </div>
  );
}

// Intermediate component
function ParentComponent({ data }) {
  return (
    <div>
      <ChildComponent data={data} />
    </div>
  );
}

// Deeply nested component that actually uses the data
function ChildComponent({ data }) {
```

```
  return <div>{data}</div>;
}
```

In this example, the `data` prop is passed from the App component through the `ParentComponent down to the ChildComponent`. The ParentComponent itself doesn't use the data prop; it merely passes it down. `This process of passing data through intermediate components that don't use the data is what is referred to as prop drilling`.

Prop drilling can make the code harder to maintain, especially as the application grows and the number of components in the hierarchy increases. To mitigate this, developers often use other state management solutions, like the `React Context API`, `Redux`, or `other state management libraries`, to avoid passing props through multiple layers of components. These alternatives provide a centralized way to manage and access state without the need for prop drilling.

Q) What is `lifting the state up`?

> `Lifting state up` in React refers to the practice of `moving the state from a lower-level (child) component to a higher-level (parent or common ancestor) component in the component tree`. This is done to share and manage state across multiple components.

When a child component needs access to certain data or needs to modify the data, instead of keeping that data and the corresponding state management solely within the child component, we move the state to a shared ancestor component. By doing so, the parent component becomes the source of truth for the state, and it can pass down the necessary data and functions as props to its child components.

💡 Here's a simple example to illustrate `lifting state up`:

```
// Parent component
class ParentComponent extends React.Component {
```

```
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  incrementCount = () => {
    this.setState((prevState) => ({
      count: prevState.count + 1,
    }));
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <ChildComponent count={this.state.count} onIncrement=
{this.incrementCount} />
      </div>
    );
  }
}

// Child component
function ChildComponent({ count, onIncrement }) {
  return (
    <div>
      <p>Child Count: {count}</p>
      <button onClick={onIncrement}>Increment</button>
    </div>
  );
}
```

In this example, the ParentComponent holds the state (count), and it passes both the state value (count) and a function (onIncrement) down to the ChildComponent

as props. The child component can then display the count and trigger an increment when the button is clicked.

By lifting the state up to a common ancestor, you centralize the state management, making it easier to control and share state among components. This pattern is especially useful in larger React applications where multiple components need access to the same data or where the state needs to be synchronized across different parts of the application.

🔗*Lifting Stateup*

Q) What are `Context Provider` and `Context Consumer` ?

> In React, the `Context API` provides a `way to pass data through the component tree without having to pass props manually at every level`. The two main components associated with the `Context API are the Context Provider and Context Consumer`.

> `Context Provider`: The Context Provider is a `component that allows its children to subscribe to a context's changes`. It accepts a value prop, which is the data that will be shared with the components that are descendants of this provider. The Provider component is created using `React.createContext()` and then rendered as part of the component tree. It establishes the context and provides the data to its descendants.

💡 Here's an example:

```
// Creating a context
const MyContext = React.createContext();
```

```
// Parent component serving as the provider
class MyProvider extends React.Component {
  state = {
    data: "Hello from Context!",
  };

  render() {
    return (
      <MyContext.Provider value={this.state.data}>
        {this.props.children}
      </MyContext.Provider>
    );
  }
}
```

`Context Consumer`: The Context Consumer is a component that subscribes to the changes in the context provided by its nearest Context Provider ancestor. It allows components to access the context data without the need for prop drilling. The Consumer component is used within the JSX of a component to consume the context data. It takes a function as its child, and that function receives the current context value as an argument. Here's an example:

```
// Child component consuming the context
class MyConsumerComponent extends React.Component {
  render() {
    return (
      <MyContext.Consumer>
        {(contextData) => (
          <p>{contextData}</p>
        )}
      </MyContext.Consumer>
    );
  }
}
```

By using the Context Provider and Context Consumer, you can avoid prop drilling and make it easier to share global or shared state across different parts of your React application. This is particularly useful when passing data to deeply nested components without explicitly passing the data through each intermediate component.

Q) If we `don't pass a value to the provider does it take the default value` ?

> Yes, If we don't pass a value to the Provider in React's Context API, `it does use the default value specified when creating the context using React.createContext(defaultValue)`.

💡 Here's the corrected explanation:

```
// Creating a context with a default value
const MyContext = React.createContext("Default Value");

// Parent component serving as the provider without providing
a value
class MyProvider extends React.Component {
  render() {
    return (
    <MyContext.Provider>
        {this.props.children}
      </MyContext.Provider>
    );
  }
}
```

In this example, if we don't provide a value to the `MyContext.Provider`, it will use the default value ("Default Value" in this case) specified during the creation of the

context. Any component that consumes this context using `MyContext.Consumer` will receive the default value if there is no Provider higher up the tree providing a different value.