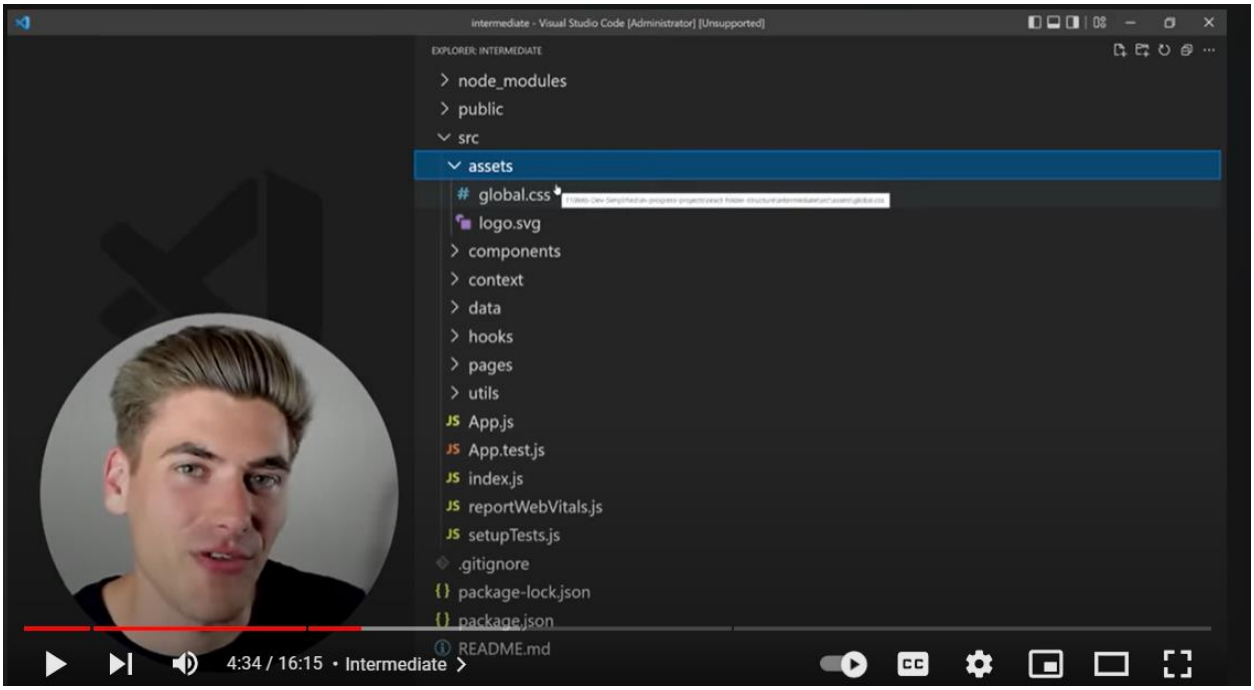
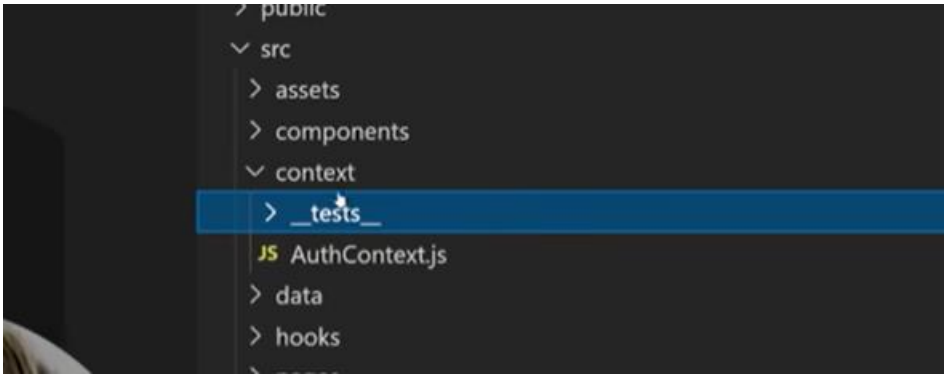


Assets – contains img or global css



Context - context provides a way to share data between components without explicitly passing it through each level of the component tree. It allows you to create a global state or share specific data between components that are not directly connected by parent-child relationships.

In the simplest terms, context in React is a way to share data between components without having to pass it through multiple levels of the component tree using props.

Imagine you have a large React application with many nested components. Usually, if you want to pass some data from a top-level component to a deeply nested component, you would have to pass that data as props through each intermediate component in the hierarchy. This is called "prop drilling," and it can become cumbersome and make the code harder to maintain.

Context solves this problem by providing a mechanism to create a shared data store that can be accessed by any component in the application, regardless of its position in the component tree. It allows you to define data at a higher level and make it available to any component that needs it, eliminating the need for passing props through multiple layers.

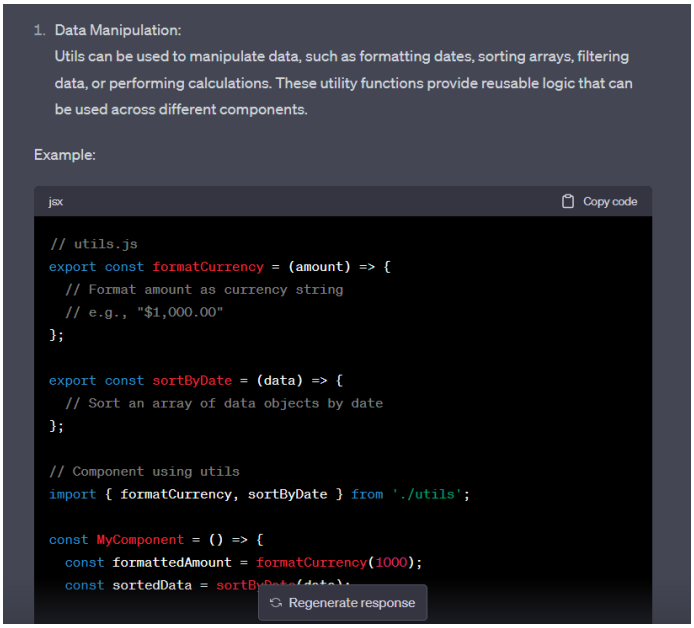
With context, you create a context object that holds the shared data, and then you wrap the components that need access to that data with a context provider component. The provider component makes the data available to its child components. Any component within the provider's descendant components can access the shared data using the context.

By using context, you can simplify the process of sharing data and avoid excessive prop drilling, making your code cleaner and more maintainable.

It's important to note that context should be used judiciously. It's best suited for cases where certain data needs to be accessed by many components throughout the application. If the data is only relevant to a small portion of the component tree, using props might still be the appropriate approach.

In summary, context is a way to create a shared data store in React that allows components to access data without the need for prop drilling. It simplifies the process of sharing data and makes the code more maintainable, especially in larger applications with complex component hierarchies.

Utils – in ReactJS, the term "utils" is short for "utilities" and generally refers to helper functions or modules that provide commonly used functionality or perform specific tasks. Utils are not specific to React itself but are often used in React projects to handle various operations.



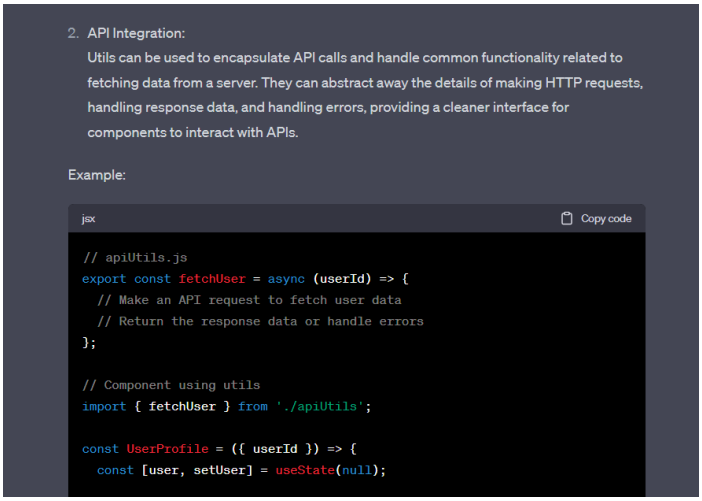
1. Data Manipulation:
   Utils can be used to manipulate data, such as formatting dates, sorting arrays, filtering data, or performing calculations. These utility functions provide reusable logic that can be used across different components.

Example:

jsx Copy code

```jsx
// utils.js
export const formatCurrency = (amount) => {
  // Format amount as currency string
  // e.g., "$1,000.00"
};

export const sortByDate = (data) => {
  // Sort an array of data objects by date
};

// Component using utils
import { formatCurrency, sortByDate } from './utils';

const MyComponent = () => {
  const formattedAmount = formatCurrency(1000);
  const sortedData = sortBy
```

Regenerate response



2. API Integration:
   Utils can be used to encapsulate API calls and handle common functionality related to fetching data from a server. They can abstract away the details of making HTTP requests, handling response data, and handling errors, providing a cleaner interface for components to interact with APIs.

Example:

jsx Copy code

```jsx
// apiUtils.js
export const fetchUser = async (userId) => {
  // Make an API request to fetch user data
  // Return the response data or handle errors
};

// Component using utils
import { fetchUser } from './apiUtils';

const UserProfile = ({ userId }) => {
  const [user, setUser] = useState(null);
```

```jsx
// apiUtils.js
export const fetchUser = async (userId) => {
  // Make an API request to fetch user data
  // Return the response data or handle errors
};

// Component using utils
import { fetchUser } from './apiUtils';

const UserProfile = ({ userId }) => {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const getUserData = async () => {
      try {
        const userData = await fetchUser(userId);
        setUser(userData);
      } catch (error) {
        // Handle error
      }
    };

    getUserData();
  }, [userId]);

  // Use the user data

  return (
```

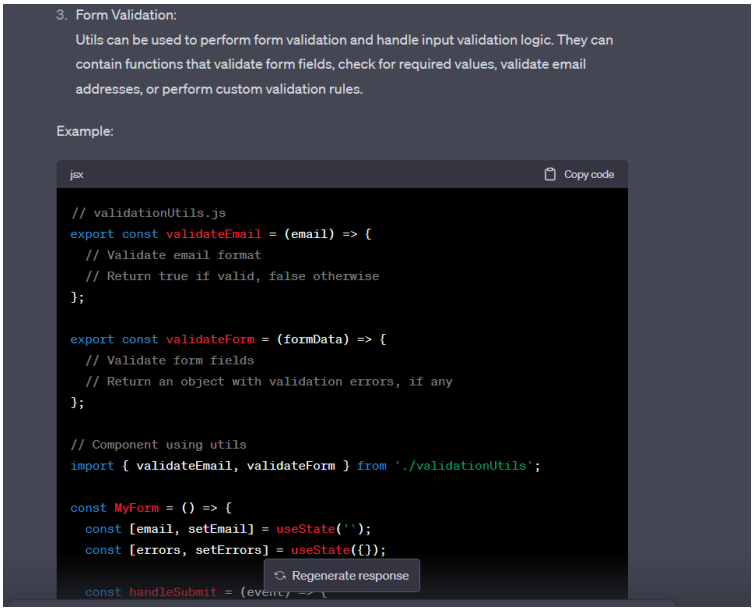
Regenerate response

Send a message.

3. Form Validation:

   Utils can be used to perform form validation and handle input validation logic. They can contain functions that validate form fields, check for required values, validate email addresses, or perform custom validation rules.

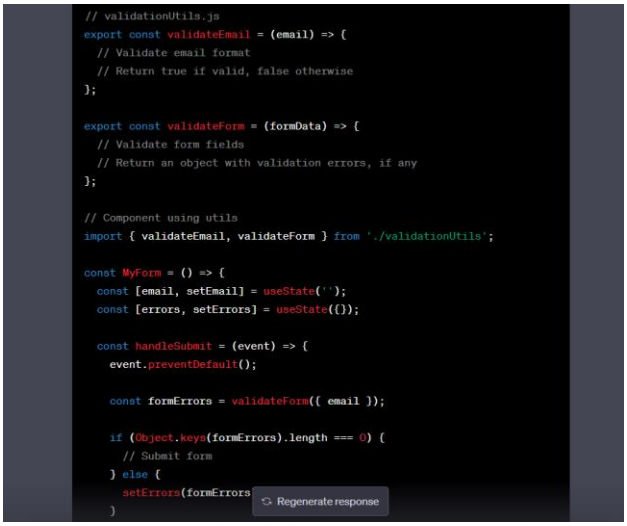
Example:



jsx — Copy code

```jsx
// validationUtils.js
export const validateEmail = (email) => {
  // Validate email format
  // Return true if valid, false otherwise
};

export const validateForm = (formData) => {
  // Validate form fields
  // Return an object with validation errors, if any
};

// Component using utils
import { validateEmail, validateForm } from './validationUtils';

const MyForm = () => {
  const [email, setEmail] = useState('');
  const [errors, setErrors] = useState({});

  const handleSubmit = (event) => {
```

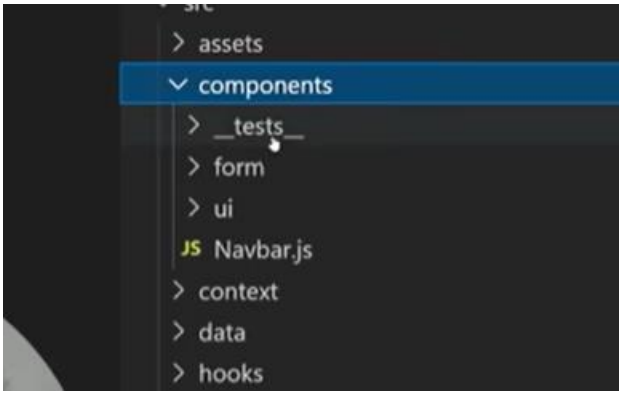
Regenerate response



```jsx
// validationUtils.js
export const validateEmail = (email) => {
  // Validate email format
  // Return true if valid, false otherwise
};

export const validateForm = (formData) => {
  // Validate form fields
  // Return an object with validation errors, if any
};

// Component using utils
import { validateEmail, validateForm } from './validationUtils';

const MyForm = () => {
  const [email, setEmail] = useState('');
  const [errors, setErrors] = useState({});

  const handleSubmit = (event) => {
    event.preventDefault();

    const formErrors = validateForm({ email });

    if (Object.keys(formErrors).length === 0) {
      // Submit form
    } else {
      setErrors(formErrors
    }
```

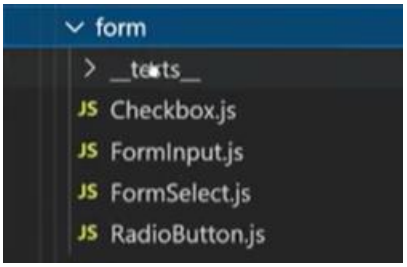
Regenerate response

Components – this folder contains the components that can be reused by other pages



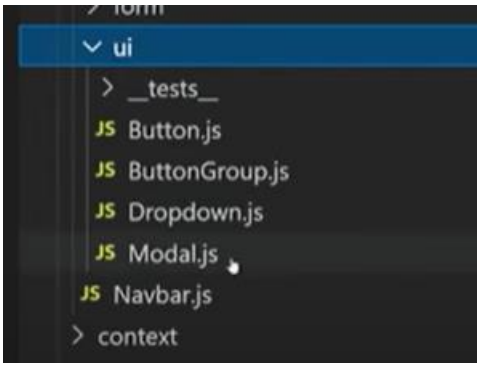
Form Components:

Form components are responsible for handling user input and managing form state. They typically include input fields, checkboxes, radio buttons, and submit buttons. Form components encapsulate the logic for capturing user input, validating the input, and submitting the form data.



UI Components:

UI components, also known as presentational or dumb components, are responsible for rendering the user interface and displaying data. They receive data and event handlers as props and provide a visual representation of the application's state.



In simpler terms, form components are responsible for the "thinking" part of a form, handling user input, and managing the form's behavior. UI components, on the other hand, are responsible for the "look and feel" part of a form, displaying the form visually and providing a pleasant user interface.

Pages – this folder contains the name page and the unique components to it